INTRODUCCIÓN A LENGUAJES DE PROGRAMACIÓN

Primera Edición

Autor: Ing. Sergio Flores Macías

Introducción a Lenguajes de Programación

Sergio Flores Macías

1ra. Edición

Autor: Sergio Eloy Flores Macías

Publicado por: Sergio Eloy Flores Macías

ISBN: 978-9942-11-321-4

Contacto: Facultad de Ingeniería en Electricidad y Computación - ESPOL, Km. 30.5 Vía Perimetral. Email: sflores@fiec.espol.edu.ec

El libro ha sido en parte construido modificando y adaptando las valiosas contribuciones de las siguientes personas que decidieron también compartir su trabajo utilizando licencias abiertas de Creative Commons:

- Anthony A. Aaby
- Mike Grant
- Zachary Palmer
- Scott Smith
- Roberto Rodríguez Echeverría
- Encarna Sosa Sánchez
- Álvaro Prieto Ramos

También se reconoce la contribución, especialmente en varios gráficos, de los repositorios libres Wikimedia, Connexions y Flickr.



Prólogo

Se han escrito muchos libros sobre Lenguajes de Programación, la mayoría desde la visión de los implementadores de estos lenguajes, pero muy pocos desde una visión más holística para el Profesional en Ciencias Computacionales. Este libro, recoge de manera única, las experiencias educativas levantadas durante más de 15 años de cátedra de la materia "Lenguajes de Programación", impartida por el Ing. Sergio Flores en la Facultad de Ingeniería en Electricidad y Computación de la Escuela Superior Politécnica del Litoral.

En los 10 capítulos de este libro, se recogen conocimientos útiles para cualquier interesado en aprender a utilizar las características de los diferentes lenguajes en el desarrollo de sistemas, así como entender el porqué se desarrollaron estas características.

El Capítulo 1 describe el porqué es importante aprender sobre Lenguajes de Programación en Ciencias Computacionales. En este capítulo se presenta además la evolución de los procesos de cómputo y como estos dieron origen a los diversos tipos de lenguaje.

El Capítulo 2 hace un recuento de la historia detallada de la evolución de los lenguajes, así como de los principales modelos de cómputo y paradigmas de programación. En este capítulo también se discute los principales criterios para el diseño de los lenguajes.

En el Capítulo 3 se realiza una revisión de las características del hardware y el software de un computador que influyen directamente en el diseño de los diversos lenguajes. Además, se presenta el concepto de máquinas virtuales, muy utilizado actualmente en lenguajes como Java o en arquitecturas de cómputo basadas en red (la Nube).

En el Capítulo 4 se desarrolla el concepto de las gramáticas formales y su vínculo inseparable con los lenguajes de programación. En este capítulo se estudian los diversos tipos de gramáticas y autómatas capaces de recorrer esas gramáticas.

En el Capítulo 5 el foco cambia a la semántica de los lenguajes y su interrelación con la sintaxis. Se revisan los diversos tipos de semántica y como cada uno de ellos nos da una perspectiva diferente del proceso de cómputo.

El Capítulo 6 presenta la teoría detrás de uno de los elementos básicos de todo lenguaje, los tipos de datos. Se discute las diferentes maneras que tiene los lenguajes para expresar, almacenar y revisar dichos tipos. Esta teoría tiene incidencia directa en la forma en que se construyen tanto los lenguajes como los programas.

En el Capítulo 7 se hace una reseña separada de las características propias de los lenguajes Orientados a Objetos, dada su relevancia actual en el desarrollo de sistemas de software. Conceptos como el de Objeto, Clase, Mensajes, Herencia y Polimorfismo son abordados desde la perspectiva de la implementación de lenguajes.

En el Capítulo 8 se trata la teoría de las Secuencias de Control y su implementación en los diversos lenguajes. Este capítulo explica el porqué de las diversas manifestaciones de estas sentencias. Se explora de manera detallada la asignación como unidad fundamental del modelo imperativo.

El Capítulo 9 desarrolla el concepto de Control de Subprogramas. En él se detalla el funcionamiento interno de un programa cuando una subrutina es llamada. Se discute además las diversas consideraciones sobre el uso de memoria y espacio en estas llamadas.

Finalmente, el Capítulo 10 presenta diversos mecanismos para la Administración del Almacenamiento. Conceptos avanzados como manejo dinámico de memoria y recolección de basura son explicados y comparados.

Estamos seguros que quienes lean este libro, o lo utilicen como parte del material de curso, tendrán una visión más completa de cómo los diversos lenguajes de programación se fueron especializando, cómo sus características fueron evolucionando y de cómo estas características y particularidades influyen en el trabajo del profesional en el desarrollo de sistemas.

Enrique Peláez

Xavier Ochoa

Prefacio

“Introducción a Lenguajes de Programación” introduce la parte teórica de un curso básico de lenguajes de programación, a estudiantes de segundo o tercer año de carrera.

La primera edición del libro está diseñada para un curso de un semestre; la teoría debe estar ligada al desarrollo de proyectos en diferentes lenguajes para que el estudiante pueda comparar la versatilidad, ventajas y desventajas de los diversos lenguajes en aplicaciones similares.

En la ESPOL, en el curso de Lenguajes de Programación se aplican cuatro lenguajes para los proyectos: Java, Ruby, SML y Squeak; los estudiantes desarrollan usualmente el mismo proyecto en los cuatro lenguajes, lo que permite comparar los diferentes paradigmas.

El libro consta de diez capítulos:

1. Introducción
2. Lenguajes de Programación
3. Arquitectura del Computador
4. Gramáticas Formales
5. Semántica
6. Valores y Tipos de Datos
7. Lenguajes Orientados a Objetos
8. Sentencias de Control

9. Control de Subprogramas
10. Administración de Almacenamiento

El estudiante encontrará en estos capítulos la teoría fundamental de los lenguajes que le permitirá desarrollar una base teórica para los siguientes cursos en el área de computación. El libro debe acompañarse de manuales de los lenguajes que se utilicen en el curso.

El desarrollo del libro ha sido realizado bajo el concepto de "Creative Commons" por lo que se lo puede copiar o modificar para utilización no comercial en base a la licencia que existe para el efecto.

El autor agradece la colaboración de los colegas Dr. Xavier Ochoa, Dr. Enrique Peláez y M.Sc. Miguel Yapur por sus comentarios, sugerencias y ayuda prestada para la elaboración de la primera edición del libro.

Finalmente, lo más importante: agradecer a mi familia por su comprensión y apoyo.

Sergio Flores

Junio 2012

Índice general

Capítulo 1

Introducción

1.1. ¿Por qué estudiar lenguajes de programación?

Las razones que motivan a estudiar nuevos lenguajes de programación son muy similares a los que motivan a aprender un nuevo lenguaje natural con una diferencia: el número de lenguajes de computación se incrementa aceleradamente por lo que es importante escoger de manera cuidadosa los lenguajes que se deben aprender y utilizar. Algunas de las razones más importantes para aprender nuevos lenguajes son:

- Mejorar la habilidad de desarrollar nuevos algoritmos. Los lenguajes se diferencian entre si por algunas características propias del tipo de lenguajes; por ejemplo, ML (Meta Lenguaje) tiene las propiedades de los lenguajes funcionales siendo una de ellas la recursividad; aprender a utilizar nuevos conceptos permite generar algoritmos utilizando estas nuevas construcciones.

- Mejorar la utilización de los lenguajes que ya conoce. Poder comparar los lenguajes conocidos, permite analizar sus similitudes y diferencias; por lo que se potencia la utilización de las construcciones existentes en un lenguaje pero que el estudiante anteriormente no las ha aplicado.

- Incrementar el vocabulario de construcciones útiles. Aprender un nuevo lenguaje permite estudiar nuevas formas de construcciones que previamente el estudiante no las conocía. Estos nuevos conceptos incrementarán el vocabulario del lector y mejorará su comunicación.

- Poder escoger mejor el lenguaje apropiado. Muchos lenguajes tienen aplicaciones muy variadas incluso no esperadas; por ejemplo, el lenguaje de sistemas C; sin embargo, para cada tipo de aplicación hay lenguajes que son más eficientes. Conocer diferentes lenguajes y su gramática permite seleccionar el lenguaje que más se adapte al problema a solucionarse.

- Facilitar el aprendizaje de nuevos lenguajes. La aparición de nuevos lenguajes y la aplicabilidad de ellos es un proceso dinámico por lo que el programador debe estar al tanto de estos desarrollos. La comunidad TIOBE genera un indicador de popularidad relativa de los lenguajes de programación; por ejemplo, en Marzo de 2012, Java, C y C# son los más populares[1].

  > "Para el que conoce dos lenguajes naturales es más fácil aprender un tercero"; en los lenguajes de programación una vez que se tiene conocimiento de los conceptos, gramáticas y sus construcciones es más sencillo, en base a comparaciones, poder aprender un nuevo lenguaje y utilizarlo eficientemente.

- Facilitar el diseño de nuevos lenguajes. El rango de diseño de lenguajes es muy amplio: desde construir un lenguaje de alto nivel hasta algo más común, como es el de diseñar aplicaciones que requieren de interfaces y protocolos de comunicación.

El conocimiento de la gramática de los lenguajes y el manejo efectivo de sus construcciones permiten tener un buen estilo de programación; Edsger Dijkstra en la introducción a su "Short Introduction to the Art of Programming":

> Es mi propósito transmitir la importancia del buen gusto y estilo de programación, [pero] los elementos de estilo presentados sirven solamente para ilustrar que beneficios pueden ser derivados del "estilo" en general. En este aspecto, me siento igual al profesor de composición de un conservatorio. Él no les enseña a sus alumnos como

[1] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html



Figura 1.1: Libros para aprender Lenguajes de Programación específicos

> componer una sinfonía en particular, él debe ayudar a sus estudiantes a encontrar su mejor estilo y debe explicarles lo que esto implica.

1.2. Historia de los Lenguajes

El desarrollo de los lenguajes a lo largo del tiempo ha estado relacionado con el desarrollo de la electrónica y con las aplicaciones que se requerían implementar, desde aplicaciones numéricas hasta sistemas expertos y autónomos. Las aplicaciones y sus lenguajes asociados las podemos definir en las siguientes áreas:

1.2.1. Aplicaciones Científicas

Las primeras computadoras fueron diseñadas para cálculos numéricos que requerían de mucha precisión. Las computadoras analógicas y las máquinas cal-

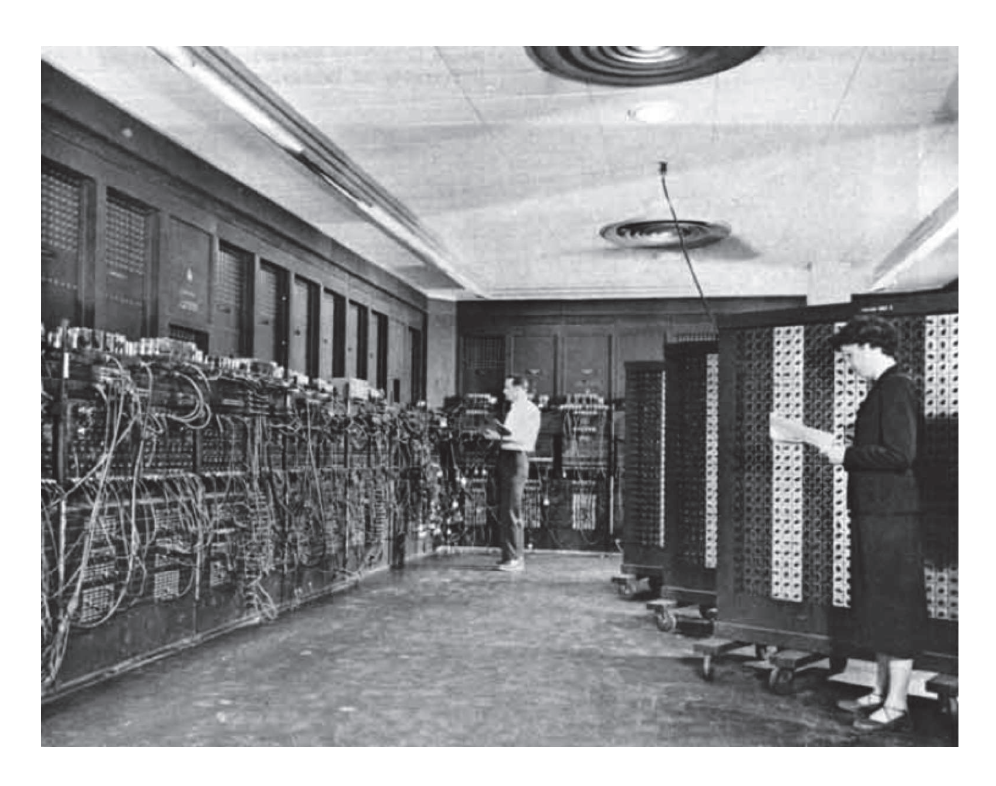


Figura 1.2: Computadora ENIAC

culadoras operadas por humanos precedieron a las computadoras digitales[2].

Zuse, Colossus y ENIAC en la década de 1940 fueron las primeras computadoras digitales programables. ENIAC fue diseñada para programas de cálculo numérico; los primeros programas de prueba fueron para el desarrollo de la bomba atómica y posteriormente para cálculos balísticos para artillería.

El primer lenguaje de alto nivel FORTRAN (FORmula TRANslating system) fue desarrollado en los años 50 para aplicaciones científicas y de ingeniería; una de las limitaciones de este lenguaje era que no permitía recursividad; FORTRAN ha continuado desarrollándose y actualmente se tiene FORTRAN 2008. La utilización de este lenguaje, principalmente en ingeniería, se debe al desarrollo de rutinas de cálculos complejos que son totalmente confiables y eficientes.

[2] http://en.wikipedia.org/wiki/History_of_computing_hardware

El dominio mundial de FORTRAN preocupó a los europeos y diseñaron un nuevo lenguaje que compitiera con FORTRAN; a mediados de los 50 se desarrolló, en el Instituto Tecnológico de Zúrich, ALGOL (ALGOrithmic Language) y su primera especificación fue ALGOL 58 y luego ALGOL 60 que fue el estándar del lenguaje. ALGOL 60 fue el primer lenguaje descrito mediante una simbología que definía la gramática del lenguaje; en esta definición intervinieron John Backus que trabajó en FORTRAN y Peter Naur editor del reporte ALGOL 60; a sugerencia de Donald Knuth este meta-lenguaje se denominó BNF (Backus-Naur Form). ALGOL tuvo mucho éxito académico por su gramática moderna orientada a algoritmos y es la raíz de muchos otros lenguajes empezando por PASCAL; sin embargo, no tuvo el éxito comercial que se esperaba de este lenguaje.

PASCAL fue desarrollado por Niklaus Wirth en 1968; la gramática introdujo nuevos conceptos entre ellos el de poder definir nuevos tipos estructurados complejos. PASCAL fue el lenguaje de aprendizaje utilizado en las universidades en la década del 70 y 80.

1.2.2. Aplicaciones de Negocios

Las aplicaciones de negocios se empezaron a desarrollar en los años 60 y requerían lenguajes que manejaran un alto volumen de datos con precisión numérica. El primer lenguaje exitoso fue COBOL (COmmon Business Oriented Language) que apareció en 1960 y hasta ahora es utilizado; COBOL 2002 incluye programación orientada a objetos.

IBM, a finales de los 60, desarrolló RPG (Report Program Generator) y RPG II. Este lenguaje también tenía lógica fija, los archivos se abrían al inicio del programa y se cerraban automáticamente al término del programa. Este lenguaje no se desarrolló mucho, después de 1970.

1.2.3. Inteligencia Artificial

Esta es un área de amplio desarrollo de cálculo simbólico. La manipulación de símbolos, en vez de números, caracteriza a las aplicaciones de Inteligencia Artificial (IA). El primer lenguaje utilizado en IA fue LISP (LISt Processing), desarrollado por John McCarthy en 1959; este lenguaje, junto con FORTRAN, son los dos lenguajes más antiguos ampliamente utilizados. LISP está basado en el concepto de cálculo lambda desarrollado por Alonzo Church y, como su

nombre lo indica, en el procesamiento de listas. LISP influyó, en esa época, en el desarrollo de nuevos lenguajes como Smalltalk y posteriormente fue la raíz del desarrollo de los lenguajes funcionales (ML, Scheme, Haskell, etc.).

Las aplicaciones de IA son muy variadas; Sistemas expertos, minería de datos, visión, robótica, máquinas de aprendizaje, reconocimiento de voz, figuras, escritura, etc.

1.2.4. Programación de Sistemas

El desarrollo de las computadoras va asociado con la programación de los sistemas necesarios para que esas computadoras operen de manera eficiente; sin embargo, el diseño del hardware ha sido más confiable que el de software. Esta diferencia en tiempo y confiabilidad ha ocasionado que los nuevos equipos no puedan utilizar, de manera inicial, toda su capacidad. La programación del sistema se la desarrollaba en el lenguaje ensamblador del procesador que se estaba utilizando. La producción acelerada de nuevos procesadores ocasionaba que los programadores de sistemas tengan que aprender a codificar en los nuevos procesadores. Brian Kernighan y Dennis Ritchie de Laboratorios Bell, entre 1969 y 1972, desarrollaron un lenguaje de nivel intermedio que permitiera una fácil compilación a diferentes lenguajes ensambladores; este lenguaje lo utilizaron para el desarrollo del sistema operativo Unix en un sistema PDP-11. El lenguaje, basado en uno denominado B, se lo denominó C. C se ha constituido en el lenguaje más ampliamente utilizado de todos los tiempos en aplicaciones muy diversas. C permite acceso de bajo nivel a la memoria por medio de punteros y remplazó a los lenguajes ensambladores.

A pesar de que, por ser un lenguaje diseñado para sistemas, tiene limitaciones en cuánto a claridad y seguridad, por su eficiencia, se lo ha utilizado en una amplia gama de aplicaciones.

Bjarne Stroustrup, de Laboratorios Bell, en 1979 amplió el lenguaje C añadiéndole características de los lenguajes de alto nivel, entre estas el concepto de clases.; este lenguaje se llamo C++ para indicar el carácter incremental de C. C++ se ha convertido en uno de los lenguajes más populares y ha influenciado el desarrollo de otros lenguajes como JAVA y C#.

1.2.5. Aplicaciones Web

El desarrollo de Internet generó nuevos requerimientos de portabilidad, seguridad y estándares internacionales. JAVA, desarrollado en 1995 por James Gosling en SUN Microsystems, fue uno de los primeros lenguajes con las características para Internet; JAVA deriva su sintaxis principalmente de C++, elimina herencia múltiple y compila a un lenguaje intermedio "Bytecode" a ser interpretado en una pequeña máquina virtual "JVM" dependiente del procesador y que puede estar en el computador cliente y en su navegador.

JAVA, como otros lenguajes y máquinas, fue inicialmente diseñado para otro tipo de aplicación: programación de televisión interactiva y se lo llamó Oak, después Green, como este lenguaje no se lo pudo utilizar se lo rediseñó, y por el café Java, se lo denominó JAVA.

La utilización de HTML, con su programación embebida, en la web y la generación de contenido dinámico en Internet han facilitado la creación de nuevos lenguajes "script" como Javascript y la utilización de antiguos como Perl o generales como Ruby.

Capítulo 2

Lenguajes de Programación

2.1. Modelos de Cómputo y Clasificación de los Lenguajes

Hay tres modelos computacionales básicos, - imperativos, funcionales y lógicos, que definen los diferentes tipos de lenguajes de programación. Además de una serie de valores y operaciones asociadas, cada uno de estos modelos computacionales tiene un conjunto de operaciones que se utilizan para definir como se lleva a cabo el cómputo. El modelo funcional utiliza la aplicación de funciones, el modelo lógico utiliza la inferencia lógica y el modelo imperativo utiliza secuencias de cambios de estado. Adicionalmente, hay dos técnicas de programación o paradigmas de programación: programación concurrente y programación orientada a objetos. Estos paradigmas no son modelos computacionales, pero son tan influyentes que se igualan a estos en importancia y definen sus propios tipos de lenguaje. Un lenguaje puede pertenecer a más de un modelo o paradigma, pero usualmente se lo incluye en el que tiene más afinidad conceptual.

2.1.1. Lenguajes Imperativos

El modelo computacional imperativo consiste en un estado y las operaciones de asignación que se utilizan para modificar dicho estado. El estado es el conjunto

S_{0}-S_{0} -> S_{1} - ... -> S_{n-1}-E_{n-1} -> S_{n}

Figura 2.1: Secuencia de Estados

de pares nombre-valor de las constantes y variables. Los programas consisten en una secuencia de comandos y el cómputo consiste en el cambio del estado. Cada paso en el cálculo es el resultado de una operación de asignación.

El modelo imperativo o procedimental es importante porque modela el cambio y los cambios son una parte integral del ambiente de cómputo. Es el modelo de cómputo que está más cerca del hardware en el que los programas se ejecutan. Su cercanía a los equipos hace que sea más fácil de implementar y los programas imperativos tienden a una menor demanda de recursos del sistema (tiempo y memoria). La principal preocupación en la programación imperativa es la definición de una secuencia de cambios de estado.

Los lenguajes imperativos o procedimentales se basan en sentencias que expresan una acción o cambio de estado del programa; las sentencias se ejecutan secuencialmente y esa ejecución puede ser alterada por instrucciones de control. Los programas son construidos en base a procedimientos (subrutinas o funciones). La metodología de programación más utilizada es la programación estructurada (de arriba hacia abajo o viceversa). Los lenguajes ensambladores fueron los primeros lenguajes imperativos, FORTRAN, ALGOL, PASCAL, C están incluidos en este modelo.

2.1.2. Lenguajes Funcionales

El modelo computacional funcional consiste de un conjunto de valores, funciones y la operación de aplicación de dichas funciones. Las funciones pueden tener un nombre y pueden ser la composición de otras funciones. Las funciones pueden tomar a otras funciones como argumentos y retornar una función como resultado. Los programas consisten en la definición de las funciones y la computación consiste en la aplicación de dichas funciones a valores. El modelo funcional es importante porque ha estado en desarrollo por cientos de años y su notación y métodos forman la base sobre la cual descansan gran cantidad de metodologías para la resolución de problemas.

Los lenguajes funcionales o aplicativos toman en consideración la evaluación de la función que el programa representa más que el cambio de estado. Los lenguajes funcionales están basados en el cálculo lambda (cálculo-λ) desarrollado en 1930 con la tesis de su capacidad de computar todas las funciones computables. La diferencia básica entre las funciones de los lenguajes imperativos y los funcionales es que en estos últimos no existen efectos colaterales. El resultado depende solamente de los argumentos utilizados en la función. Esta característica es importante en el diseño de lenguajes concurrentes. La utilización de la recursividad es una de las características de la programación funcional. El resultado de un programa consiste en la aplicación de varias funciones que de manera sucesiva mapean los datos. La sintaxis de un lenguaje es similar a:

$$\text{función}_n(\ldots\text{función}_2(\text{función}_1(\text{argumentos}))\ldots)$$

El primer lenguaje funcional fue LISP; otros lenguajes funcionales son SML, Erlang, Scheme, OCaml, Haskell, F#.

2.1.3. Lenguajes Basados en Reglas

El modelo computacional lógico se basa en relaciones e inferencia lógica. Los programas consisten en la definición de relaciones y el cómputo en las inferencias. El modelo lógico es importante porque es la formalización del proceso de razonamiento. Está relacionado con las bases de datos relacionales y los sistemas expertos.

Los lenguajes con base en reglas son el cimiento del desarrollo para áreas importantes en inteligencia artificial y bases de datos tales como bases de datos activas, bases de datos deductivas y sistemas expertos. Los sistemas expertos son en la actualidad ampliamente utilizados en negocios, salud, acuicultura, juegos, entrenamiento, etc.

Los programas se basan en máquinas de inferencia que actúan sobre un conjunto de reglas base generadas por expertos en el área de aplicación. Los lenguajes más conocidos son Prolog y Datalog.

2.1.4. Lenguajes Orientados a Objetos

La utilización de la programación estructurada tiene como concepto el desarrollo del programacon base en acciones; el verbo que representaba la acción estaba

implementado en un comando o función. Los datos quedaron desprotegidos lo que generaba errores difíciles de detectar en el desarrollo de sistemas grandes, implementados por varios grupos de programadores. Se utilizaron conceptos como el de tipo de datos abstractos (ADT) para mitigar esta problemática, pero al ser discrecional no tuvo el objetivo deseado. Se requería un cambio de paradigma que protegiera la información y en los 90 se empezó a utilizar la programación orientada a objetos (POO).

Este paradigma no era nuevo; se inició en los 70 con Smalltalk, desarrollado en Xerox Parc por Alan Kay, Dan Ingalls y Adele Goldberg. El concepto básico de POO es el objeto, que es una instancia de una clase. No se puede tener acceso al objeto directamente, el objeto recibe mensajes implementados en métodos. La siguiente expresión:

$$2 + 3$$

indica que el objeto 2 recibe el mensaje "+" y como argumento el objeto 3; este mensaje es implementado con un método de suma de enteros. La programación orientada a objetos es, en la actualidad, el paradigma más utilizado; los lenguajes más populares: Java, C# y C++ pertenecen a este paradigma.

2.2. Criterios para Diseñar un Buen Lenguaje

Hoare en 1973 en su memo AIM-224[1] "Hints on Programming Language Design " indicaba:

> ... los criterios objetivos para un buen diseño de lenguaje pueden ser resumidos en cinco eslóganes: simplicidad, seguridad, traducción rápida, código objeto eficiente y legibilidad.

Los criterios de Hoare se mantienen de manera general; sin embargo, el desarrollo tanto del hardware como el de nuevos algoritmos para el diseño de lenguajes ha modificado la importancia relativa de estos criterios. Las características, entonces, de un buen lenguaje que se plantean como las más importantes en la actualidad son:

[1] ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/73/403/CS-TR-73-403.pdf

2.2.1. Legibilidad

Antes de 1970, el obtener código objeto de manera eficiente, por el costo de memoria, era la característica principal de un lenguaje; ahora, con la disminución de costos de los elementos electrónicos, una de las características principales de un lenguaje es la facilidad con la que se lo puede leer y entender. El desarrollo de conceptos relacionados al ciclo de vida de los sistemas enfatiza la importancia de la legibilidad en la disminución de los costos de mantenimiento de un sistema.

La sintaxis del lenguaje debe ser simple y clara para que sea legible; sin embargo, las construcciones, mientras más simples, son menos claras (programas en APL). Un lenguaje para ser claro requiere de multiplicidad que consiste en tener más de una construcción para realizar una operación específica; Ruby tiene esta característica. Los lenguajes ensambladores en cambio son usualmente simples.

2.2.2. Ortogonalidad

La ortogonalidad en un lenguaje es la propiedad por la cual las construcciones primitivas del mismo puedan ser combinadas de todas las maneras posibles, desarrollando nuevas construcciones y estructuras válidas. No existe lenguaje cien por ciento ortogonal, siempre hay excepciones en las nuevas construcciones. Por ejemplo, en C se pueden retornar estructuras pero no arreglos en funciones; un elemento de un arreglo puede ser de cualquier tipo excepto void; un arreglo se puede retornar si está dentro de una estructura.

El exceso de ortogonalidad también causa problemas: en C, expresiones lógicas y aritméticas pueden mezclarse, las sentencias retornan valores por lo que pueden ser utilizadas en expresiones causando confusión por la complejidad innecesaria. Lenguajes funcionales como ML tienen una buena combinación de simplicidad y ortogonalidad.

2.2.3. Naturaleza de la Aplicación

Los lenguajes de programación, si bien son considerados universales, tienen un diseño que está más orientado a cierto tipo de aplicaciones; FORTRAN fue diseñado para traducir directamente en el lenguaje fórmulas matemáticas. La implementación del algoritmo de la aplicación será más eficiente en un lenguaje con construcciones que reflejen la estructura del algoritmo.

2.2.4. Soporte para Abstracción

Abstracción es una de las características más importantes de los lenguajes modernos. La abstracción representa el significado (semántica) del objeto; la implementación es definida posteriormente. La abstracción puede tener diferentes niveles de representación. El concepto de pila puede ser representado de manera abstracta y posteriormente se la implementa como una pila de enteros, reales, cadenas, etc. Los operadores y la implementación de la estructura es otro nivel de abstracción, puede ser por ejemplo un arreglo o una lista.

Existen dos tipos de abstracciones, de control y de datos; la abstracción de control esta relacionado con la utilización de subprogramas y sus flujos de control, la abstracción de datos involucra el manejo de los bits de datos de una manera apropiada.

El lenguaje debe, en su diseño, tener el soporte necesario que le permita un manejo natural de las abstracciones.

2.2.5. Confiabilidad

Un programa es confiable si funciona de acuerdo a sus especificaciones en cualquier circunstancia. La confiablidad de un programa es un tema de extrema importancia, especialmente en sistemas que corren en tiempo real. Los errores de software han ocasionado pérdidas humanas; en una máquina de rayos X controlada por computador, errores en el programa causaron que la máquina diera dosis letales durante un tratamiento. El Arianne-5 se destruyó en su lanzamiento debido a un mal manejo de sobrecarga de números en punto flotante en el programa escrito en ADA[2] . La calidad del programa y su confiabilidad dependen de las características del lenguaje y de la exactitud de la programación.

El diseño de un lenguaje para que sea más confiable debe tener como principios: legibilidad, portabilidad, sintaxis simple, manejo de excepciones, soporte para verificación, reutilización de código, modularidad, documentación embebida en el software, estilo y estándares de programación, de los cuales se desarrollán los siguientes:

[2] http://web.cecs.pdx.edu/~harry/musings/RelLang.pdf

2.2.5.1. Portabilidad

La portabilidad se la entiende como la capacidad de mover el código a distintas plataformas de hardware y software diferentes a la que se utilizó para su implementación. Usualmente los lenguajes compilan a un lenguaje intermedio, independiente del hardware y software, y lo ejecutan en una máquina virtual dependiente de la plataforma a ser utilizada. El primer lenguaje de alto nivel que utilizó este concepto fue PASCAL, compilado a un lenguaje denominado P que ejecutaba en una máquina virtual de pila ("stack machine"); Smalltalk, Java, Ruby son algunos de los lenguajes que tienen esta característica.

2.2.5.2. Sintaxis Simple

La sintaxis compleja conlleva confusión y errores. Un programa legible debe tener una sintaxis sencilla que permita a un humano rápidamente entender un programa. Lenguajes que tienen una gramática LL(1)[3] usualmente tienen una sintaxis sencilla.

2.2.5.3. Manejo de Excepciones

La capacidad de un lenguaje de poder capturar y manejar los errores, del sistema y del programa, que se generan durante la ejecución evitando que el programa se pare, contribuye a la confiabilidad. Ada, C++, C#, Java, Ruby, ML tienen un buen manejo de excepciones. Lenguajes más antiguos como FORTRAN y C no tienen mucha capacidad de manejo de errores.

2.2.5.4. Soporte para Verificación

La verificación de que un programa es correcto mediante métodos y pruebas formales es todavía un área de desarrollo investigativo. El lenguaje, el compilador y el entorno integrado de desarrollo (IDE) deben facilitar el chequeo y prueba del programa durante el diseño, compilación y ejecución del mismo. "assert" en C y C++ y "junit" en Java son ejemplos de construcciones que soportan pruebas.

[3] Ver sección 4.4.2.1

2.2.6. Costos

Los costos de un sistema están asociados al ciclo de vida del mismo: requerimientos, diseño, implementación, verificación y mantenimiento. Los costos de mantenimiento en 1979 representaban el 67 % del costo total del software, en el 2000 representaban más del 90 %[4]. Se estima que más del 80 % del esfuerzo de mantenimiento es para acciones no correctivas[5], es decir, modificaciones.

El desarrollo del sistema debe, para reducir costos, tener un diseño modular, débilmente acoplado y confiable. El lenguaje que se utilice debe ofrecer las facilidades no solo para el diseño e implementación, sino, también para el mantenimiento a lo largo de la vida del sistema.

2.3. Evolución de los Lenguajes

2.3.1. Antes de 1940

Los primeros lenguajes de programación son anteriores a la computadora moderna. En un primer momento, los lenguajes eran códigos.

El telar de Jacquard, inventado en 1801, utiliza los agujeros en tarjetas perforadas para representar los movimientos del brazo de un telar con el fin de generar patrones decorativos de forma automática.

Durante un período de nueve meses en 1842-1843, Ada Lovelace tradujo el libro de memorias del matemático italiano Luigi Menabrea acerca de la máquina analítica de Charles Babbage. Con el artículo, se añade un conjunto de notas que especifican con todo detalle un método para calcular los números de Bernoulli con el motor, reconocido por algunos historiadores como el primer programa informático del mundo.

Herman Hollerith se dio cuenta de que podía codificar la información en tarjetas perforadas, cuando observó que los conductores de trenes codificaban la presencia de los viajeros en la posición donde perforaban sus tickets. Hollerith luego codificó los datos del censo de 1890 en tarjetas perforadas.

[4]http://users.jyu.fi/~koskinen/smcosts.htm

[5]Pigoski, Thomas M. (1996). Practical Software Maintenance. New York: John Wiley & Sons. ISBN 978-0-471-17001-3.

Los primeros códigos informáticos estaban especializados para sus aplicaciones. En las primeras décadas del siglo 20, los cálculos numéricos se basan en números decimales. Eventualmente se descubrió que la lógica se podría representar con números, no sólo con palabras. Por ejemplo, Alonzo Church fue capaz de expresar el cálculo lambda a manera de fórmula. La máquina de Turing era una abstracción de la operación de una máquina de cinta que utilizaban las compañías telefónicas. La máquina de Turing sentó las bases para el almacenamiento de los programas como datos en la arquitectura de von Neumann, mediante la representación de una máquina a través de un número finito. Sin embargo, a diferencia del cálculo lambda, el código de Turing no sirve bien como una base para lenguajes de alto nivel. Su uso principal es en el análisis riguroso de complejidad algorítmica.

Al igual que muchos "primeros" en la historia, el primer lenguaje de programación moderno es difícil de identificar. Desde el principio, las restricciones del hardware definían al lenguaje. Las tarjetas perforadas permitían 80 columnas, pero algunas de las columnas tenían que ser utilizadas para un número de clasificación en cada tarjeta. FORTRAN incluye algunas palabras clave que son las mismas que las palabras en inglés, como "IF" (si), "GOTO" (ir a) y "CONTINUE" (continuar). El uso de un tambor magnético para la memoria significaba que los programas del computador también tenían que ser intercalados con las rotaciones del tambor. Así, los programas eran más dependientes del hardware.

Para algunas personas, el concepto de "lenguaje de programación" está ligado con la potencia y la legibilidad humana. Las Máquinas de Jacquard y Charles Babbage tenían lenguajes simples, muy limitados para describir solamente las acciones que deben llevar a cabo estas máquinas. Se puede incluso considerar a las perforaciones en una pianola como un lenguaje de dominio específico.

2.3.2. La década de 1940

Las primeras máquinas, reconocibles como modernas y que funcionaban con energía eléctrica fueron creadas en la década de 1940. La velocidad limitada y la capacidad de memoria obligaron a los programadores a escribir programas a mano en lenguaje ensamblador. Pronto se descubrió que la programación en lenguaje ensamblador requería un gran esfuerzo intelectual y era propenso a errores.

En 1948, Konrad Zuse publicó un artículo acerca de su lenguaje de programación Plankalkül. Sin embargo, éste no se construyó durante su vida y su contribución

original estuvo aislada de otros desarrollos.

Algunos lenguajes importantes que se desarrollaron en este período son:

- 1943 - Plankalkül (Konrad Zuse), diseñado, pero no implementado por más de medio siglo.
- 1943 - Sistema de Codificación ENIAC, conjunto de códigos, específicos del equipo, que aparecen en 1948.
- 1949 - 1954 - Un conjunto de instrucciones mnemónicas específicas para diversos sistemas, como el ENIAC, comenzando en 1949 con C-10 para BINAC (que más tarde se convirtió en el UNIVAC). Cada juego de códigos, o conjunto de instrucciones, se adaptó a un fabricante específico.

2.3.3. Los años 1950 y 1960

En la década de 1950, los tres primeros lenguajes de programación modernos cuyos descendientes todavía están en uso generalizado hoy en día fueron diseñados:

- FORTRAN (1955), inventado por John Backus et al.;
- LISP (1958), el "LISt Processor", inventado por John McCarthy et al.;
- COBOL (1959), creado por el Comité de Corto Alcance, muy influenciado por Grace Hopper.

Otro hito en la década de 1950 fue la publicación, por un comité de científicos de computación de América y Europa, de "un nuevo lenguaje para los algoritmos", el informe ALGOL 60 ("ALGOrithmic Language") consolidó muchas ideas que circulaban en el momento y contó con dos innovaciones claves del lenguaje:

- estructura de bloques anidados: secuencias de código y las declaraciones correspondientes se podían agrupar en bloques, sin tener que convertirse en procedimientos separados, nombrados de manera explícita;
- rango léxico: un bloque puede tener sus propias variables privadas, procedimientos y funciones, invisibles para el código fuera de ese bloque, es decir, la ocultación de información.

Otra de las novedades estaba en cómo se escribe el lenguaje:

- una notación matemática exacta, Backus-Naur Form (BNF), fue utilizada para describir la sintaxis del lenguaje. Casi todos los lenguajes de programación posteriores han utilizado una variante del BNF para describir el contexto libre de parte de su sintaxis.

Algol 60 fue especialmente influyente en el diseño de lenguajes posteriores, algunos de ellos se hicieron más populares. Los grandes sistemas de Burroughs fueron diseñados para ser programados en un subconjunto extendido de Algol.

Las ideas claves de Algol fueron continuadas en la producción de ALGOL 68:

- La sintaxis y la semántica se hicieron aún más ortogonales, con rutinas anónimas, sistemas de tipificación recursiva con funciones de orden superior, etc.
- no sólo la parte libre de contexto, pero la sintaxis del lenguaje y la semántica completa se definió formalmente, en términos de la gramática de Van Wijngaarden, un formalismo diseñado específicamente para este propósito.

Algol 68 tuvo muchas características de poco uso del lenguaje (por ejemplo, bloques simultáneos y paralelos) y su complejo sistema de accesos directos sintácticos y coerciones automáticas lo hizo impopular entre los programadores y se ganó una reputación de ser difícil. Niklaus Wirth en realidad salió de la comisión de diseño para crear un lenguaje más sencillo: Pascal.

Algunos lenguajes importantes que se desarrollaron en este período incluyen:

- 1952 - Autocode
- 1954 - IPL (precursor de LISP)
- 1955 - FLOW-MATIC (precursor de COBOL)
- 1957 - FORTRAN (primer compilador)
- 1957 - COMTRAN (precursor de COBOL)
- 1958 - LISP
- 1958 - ALGOL 58

- 1959 - HECHO (precursor de COBOL)
- 1959 - COBOL
- 1959 - RPG
- 1962 - APL
- 1962 - Simula
- 1962 - SNOBOL
- 1963 - CPL (precursor de C)
- 1964 - BASIC
- 1964 - PL / I
- 1967 - BCPL (precursor de C)

2.3.4. 1967-1978: el establecimiento de paradigmas fundamentales

El período comprendido entre los años 1960 hasta fines de 1970 trajo un gran florecimiento de lenguajes de programación. La mayoría de los paradigmas más importantes de los lenguajes actualmente en uso fueron inventados en este período:

- Simula, inventado en la década de 1960 por Nygaard y Dahl como un superconjunto de Algol 60, fue el primer lenguaje diseñado para apoyar la programación orientada a objetos.
- C, uno de los primeros lenguajes de programación de sistemas, fue desarrollado por Dennis Ritchie y Ken Thompson en los Laboratorios Bell entre 1969 y 1973.
- Smalltalk (mediados de 1970) proporcionó el diseño de un lenguaje orientado a objetos.
- Prolog, diseñado en 1972 por Colmerauer, Roussel, y Kowalski, fue el primer lenguaje de programación lógico.

- ML construido como un sistema de tipo polimórfico (inventado por Robin Milner en 1973) basado en Lisp, pionero en tipos estáticos en lenguajes de programación funcional.

Cada uno de estos lenguajes dio lugar a toda una familia de descendientes, y la mayoría de los lenguajes modernos cuentan con al menos uno de ellos en su ascendencia.

En los años 1960 y 1970 también se suscitó un debate considerable sobre los méritos de la "programación estructurada", que esencialmente significaba programación sin el uso de GOTO. Este debate estaba estrechamente relacionado con el diseño del lenguaje: algunos lenguajes no incluían GO, lo que obligó a utilizar programación estructurada por los programadores. Aunque se recrudecía el debate acalorado en el momento, casi todos los programadores están de acuerdo en que, incluso en lenguajes que ofrecen GOTO, es mal estilo de programación para usarlo excepto en raras circunstancias. Como resultado, las generaciones posteriores de diseñadores de lenguajes han encontrado el debate de la programación estructurada tedioso y desconcertante.

Algunos lenguajes importantes que se desarrollaron en este período incluyen:

- 1968 - Logo
- 1969 - B (precursor de C)
- 1970 - Pascal
- 1970 - Forth
- 1972 - C
- 1972 - Smalltalk
- 1972 - Prolog
- 1973 - ML
- 1975 - Scheme
- 1978 - SQL (inicialmente sólo un lenguaje de consulta, posteriormente se amplió con construcciones de programación)

2.3.5. La década de 1980: la consolidación, los módulos, el rendimiento

La década de 1980 fue de relativa consolidación de los lenguajes imperativos. En vez de inventar nuevos paradigmas, los nuevos lenguajes fueron elaborados a partir de las ideas desarrolladas en la década anterior. C++ combinó orientación a objetos y programación de sistemas. El gobierno de Estados Unidos estandarizó Ada, un lenguaje de programación de sistemas destinados a ser utilizados por los contratistas del departamento de defensa. En Japón y en otros lugares, grandes sumas de dinero fueron gastadas investigando los llamados lenguajes de programación de quinta generación, que incorporan construcciones de programación lógica. La comunidad de diseñadores de los lenguajes funcionales estandarizó ML y Lisp. La investigación en Miranda, un lenguaje funcional con evaluación perezosa, comenzó a tomar fuerza en esta década.

Una tendencia nueva e importante en el diseño de lenguajes era una mayor atención a la programación de sistemas a gran escala a través del uso de módulos, o grandes unidades de organización de código. Modula, Ada, ML y todos los sistemas de módulos importantes fueron desarrollados en la década de 1980. Los sistemas modulares se basaron a menudo en las construcciones de programación genéricas que son, en esencia, los módulos parametrizados[6].

A pesar de que no aparecieron nuevos paradigmas importantes de lenguajes de programación imperativos, muchos investigadores se expandieron en las ideas de los lenguajes anteriores y los adaptaron a los nuevos contextos. Por ejemplo, los lenguajes Argus y Esmeralda son sistemas adaptados de programación orientada a objetos a sistemas distribuidos.

La década de 1980 también trajo avances en la implementación de los lenguajes de programación. El movimiento RISC en la arquitectura de computadores postuló que el hardware debe ser diseñado para compiladores y no para los programadores. Con la ayuda del incremento en la velocidad del procesador se permitieron técnicas de compilación cada vez más agresivas, el movimiento RISC suscitó un mayor interés en la tecnología de compilación de los lenguajes de alto nivel.

El desarrollo de tecnologías de los lenguajes continuó en esta línea hasta bien entrada la década de 1990.

Algunos lenguajes importantes que se desarrollaron en este período incluyen:

[6] Ver sección 7.7 Polimorfismo

- 1980 - C++ (como C con clases, el nombre cambió en julio de 1983)
- 1983 - Ada
- 1984 - Common Lisp
- 1984 - MATLAB
- 1985 - Eiffel
- 1986 - Objective-C
- 1986 - Erlang
- 1987 - Perl
- 1988 - Tcl
- 1988 - Mathematica
- 1989 - FL (Backus)

2.3.6. La década de 1990: la era de Internet

El rápido crecimiento de Internet en la década de 1990 fue el siguiente gran acontecimiento histórico en los lenguajes de programación. Con la apertura de una plataforma totalmente nueva para los sistemas informáticos, Internet crea una oportunidad para que se adopten los nuevos lenguajes. En particular, el lenguaje de programación Java se hizo popular debido a su pronta integración con el navegador web Netscape Navigator, y varios lenguajes de scripting, alcanzado un uso generalizado en el desarrollo de aplicaciones personalizadas para servidores web.

La década de 1990 no vio ninguna novedad fundamental de los lenguajes imperativos, pero si la recombinación y la maduración de las viejas ideas. Esta era comenzó la difusión de los lenguajes funcionales. Una filosofía de importancia era la productividad del programador. Surgieron muchos lenguajes de "Desarrollo Rápido de Aplicaciones" ("Rapid Aplication Development"), que por lo general venían con un IDE, recolección de basura, y eran descendientes de los lenguajes anteriores. Todos estos lenguajes eran orientados a objetos. Estos incluyen Object Pascal, Visual Basic y Java. Java, en particular, recibió mucha atención. Más radicales e innovadores que los lenguajes de RAD fueron

los nuevos lenguajes de scripting. Estos no descendieron directamente de otros lenguajes y ofrecieron nuevas sintaxis y la incorporación de características más liberales. Muchos consideran que estos lenguajes de programación pueden ser más productivos que incluso los lenguajes de RAD, pero a menudo debido a las opciones los programas construidos de pequeños programas más simples, son más difíciles de escribir y mantener. Sin embargo, los lenguajes de secuencias de comandos llegaron a ser los más usados en conexión con la web.

Algunos lenguajes importantes que se desarrollaron en este período incluyen:

- 1990 - Haskell
- 1991 - Python
- 1991 - Visual Basic
- 1991 - HTML
- 1993 - Ruby
- 1993 - Lua
- 1994 - CLOS (parte de la norma ANSI Common Lisp)
- 1995 - Java
- 1995 - Delphi (Object Pascal)
- 1995 - JavaScript
- 1995 - PHP
- 1996 - WebDNA
- 1997 - Rebol
- 1999 - D

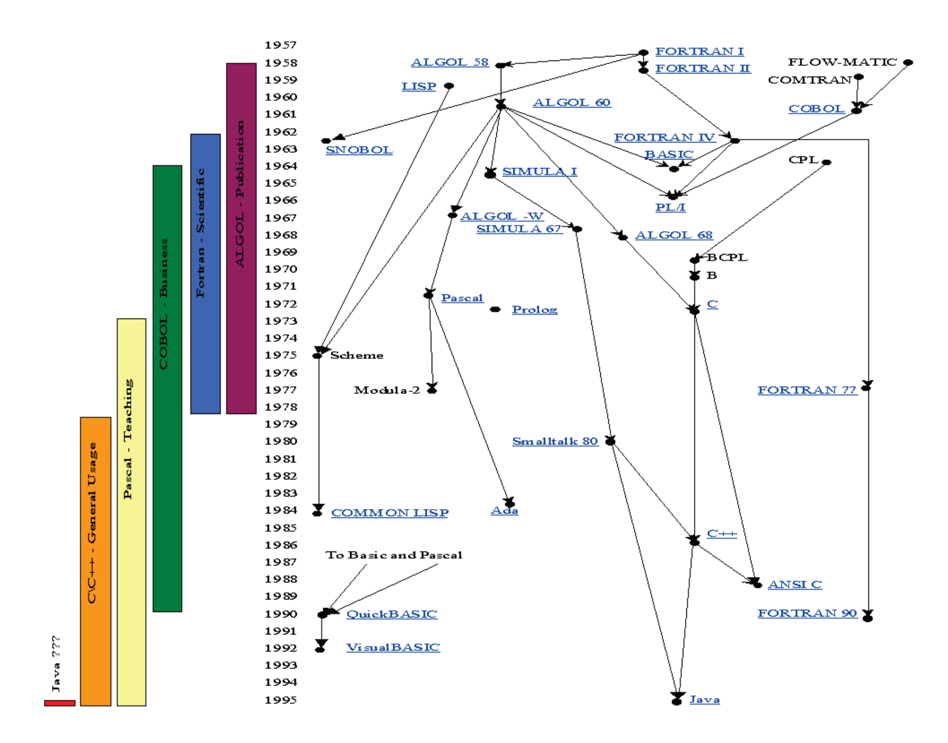


Figura 2.2: Evolución de los lenguajes de programación

2.3.7. Las Tendencias Actuales

La evolución de los lenguajes de programación continúa, tanto en la industria como en la investigación. Algunas de las tendencias actuales incluyen:

- Construcciones para apoyar la programación concurrente y distribuida.
- Mecanismos para la adición de verificación de seguridad y fiabilidad de los lenguajes: la comprobación extendida estática, control de flujo de información y seguridad estática de los subprocesos.
- Mecanismos alternativos para la modularidad: mixins, delegados, programación orientada a aspectos ("Aspect Oriented Programming").
- Orientación a componentes de desarrollo de software.

- Metaprogramación, reflexión o acceso al árbol de sintaxis abstracta.
- Mayor énfasis en la distribución y la movilidad.
- Integración con bases de datos, incluyendo XML y bases de datos relacionales.
- Soporte para Unicode, para que el código fuente (el texto del programa) no se limite a los caracteres que figuran en el código ASCII.
- XML para la interfaz gráfica (XUL, XAM).
- Código abierto como una filosofía de desarrollo de los lenguajes, incluyendo el conjunto de compiladores de GNU y los lenguajes recientes, como Python, Rubí, y Squeak.
- Programación Orientada a Aspectos (AOP).
- Delegación o métodos más complejos de gestión de listas de paquetes de datos.
- Lenguajes paralelos para la codificación masiva de miles de unidades de procesadores gráficos GPU y operación de matrices de supercomputación como OpenCL.

Algunos lenguajes importantes desarrollados durante este período incluyen:

- 2000 - ActionScript
- 2001 - C #
- 2001 - Visual Basic NET.
- 2002 - F #
- 2003 - Groovy
- 2003 - Scala
- 2003 - Factor
- 2007 - Clojure
- 2009 - Go
- 2011 - Dart

Capítulo 3

Arquitectura del Computador

La arquitectura de las computadoras es el diseño conceptual y la estructura operacional fundamental de un sistema de computación. En el desarrollo de un lenguaje de programación, la arquitectura del hardware y del software del computador influencia el diseño de un lenguaje de dos maneras: (1) el equipo subyacente en el que los programas escritos en el lenguaje se ejecutarán, y (2) el modelo de ejecución, o la computadora virtual, que soportará ese lenguaje en el hardware real.

3.1. Arquitectura de Hardware

Es decir, es un modelo y una descripción funcional de los requerimientos y las implementaciones de diseño para varias partes de una computadora, con especial interés en la forma en que la unidad central de proceso (CPU) trabaja internamente y accede a las direcciones de memoria. También, suele definirse como la forma de seleccionar e interconectar componentes de hardware para crear computadoras según los requerimientos de funcionalidad, rendimiento y costo. El computador recibe y envía la información a través de los periféricos por medio de los canales. La CPU es la encargada de procesar la información que le llega al computador. El intercambio de información se tiene que hacer con los periféricos y la CPU. Todas aquellas unidades de un sistema exceptuando la CPU se denominan periféricos, por lo que el computador tiene dos partes bien

diferenciadas, que son: la CPU (encargada de ejecutar programas y que está compuesta por la memoria principal, la Unidad Aritmética Lógica y la Unidad de Control) y los periféricos (que pueden ser de entrada, salida, entrada-salida y comunicaciones).

La implantación de instrucciones es similar al uso de una serie de desmontaje en una fábrica de manufacturas. En las cadenas de montaje, el producto pasa a través de muchas etapas de producción antes de tener el producto terminado. Cada etapa o segmento de la cadena está especializada en un área específica de la línea de producción y lleva a cabo siempre la misma actividad. Esta tecnología es aplicada en el diseño de procesadores eficientes. A estos procesadores se les conoce como "pipeline processors". Éstos están compuestos por una lista de segmentos lineales y secuenciales en donde cada segmento lleva a cabo una tarea o un grupo de tareas computacionales. Los datos que provienen del exterior se introducen en el sistema para ser procesados. La computadora realiza operaciones con los datos que tiene almacenados en memoria, produciendo nuevos datos o información para uso externo.

Las arquitecturas y los conjuntos de instrucciones se pueden clasificar considerando los siguientes aspectos:

- Almacenamiento de operaciones en la CPU: dónde se ubican los operadores aparte de la sustractora informativa (SI).
- Número de operandos explícitos por instrucción: cuántos operandos se expresan en forma explícita en una instrucción típica. Normalmente son 0, 1, 2 y 3.
- Posición del operando: ¿Puede cualquier operando estar en memoria?, o ¿deben estar algunos o todos en los registros internos de la CPU?
- Cómo se especifica la dirección de memoria (modos de direccionamiento disponibles).
- Operaciones: Qué operaciones están disponibles en el conjunto de instrucciones.
- Tipo y tamaño de operandos y cómo se especifican.

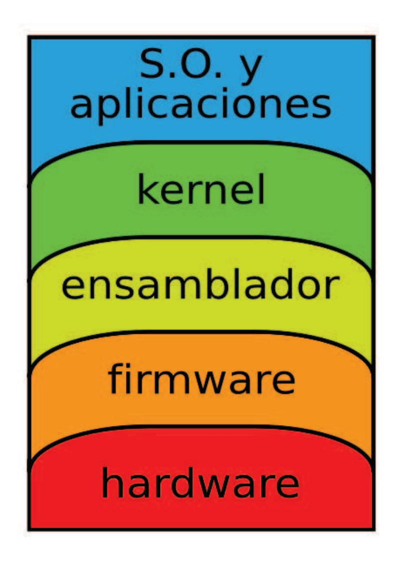


Figura 3.1: Capas de la arquitectura del computador

3.1.1. Arquitectura de von Neumann

La arquitectura de von Neumann es una familia de arquitecturas de computadoras que utilizan el mismo dispositivo de almacenamiento, tanto para las instrucciones como para los datos. La mayoría de computadoras modernas están basadas en esta arquitectura, aunque pueden incluir otros dispositivos adicionales (por ejemplo, para gestionar las interrupciones de dispositivos externos como ratón, teclado, etc.).

3.1.1.1. Origen

El nacimiento u origen de la arquitectura von Neumann surge a raíz de una colaboración en el proyecto ENIAC del matemático de origen húngaro, John von Neumann. Este trabajaba en 1945 en el Laboratorio Nacional Los Álamos cuando se encontró con uno de los constructores de la ENIAC. Compañero de Albert Einstein, Kurt Gödel y Alan Turing en Princeton, von Neumann se interesó por el problema de la necesidad de recablear la máquina para cada nueva tarea.



Figura 3.2: Foto de John von Neumann

En 1949 había encontrado y desarrollado la solución a este problema, consistente en poner la información sobre las operaciones a realizar en la misma memoria utilizada para los datos, escribiéndola de la misma forma, es decir en código binario. Su "EDVAC" fue el modelo de las computadoras de este tipo construidas a continuación. Se habla desde entonces de la arquitectura de von Neumann, aunque también diseñó otras formas de construcción. El primer computador comercial construido en esta forma fue el UNIVAC I, fabricado en 1951 por la Sperry-Rand Corporation y comprado por la Oficina del Censo de Estados Unidos.

3.1.1.2. Organización

Las computadoras con esta arquitectura constan de cinco partes: La unidad aritmético-lógica o ALU, la unidad de control, la memoria, un dispositivo de entrada/salida y las barras de dirección, de control y de datos, que proporcionan un medio de transporte de los datos entre las distintas partes. Un computador con esta arquitectura realiza o emula los siguientes pasos secuencialmente:

1. Enciende el computador y obtiene la siguiente instrucción desde la memoria en la dirección indicada por el contador de programa, transmitida por la barra de dirección, y la guarda en el registro de instrucción.

2. Aumenta el contador de programa en la longitud de la instrucción para apuntar a la siguiente instrucción.
3. Decodifica la instrucción mediante la unidad de control. Ésta se encarga de coordinar el resto de componentes del computador para realizar una función determinada.
4. Se ejecuta la instrucción. Ésta puede cambiar el valor del contador del programa, permitiendo así operaciones repetitivas. El contador puede cambiar también cuando se cumpla una cierta condición aritmética, haciendo que el computador pueda 'tomar decisiones', que pueden alcanzar cualquier grado de complejidad, mediante la aritmética y lógica anteriores.

El término arquitectura de von Neumann se acuñó a partir del memorando "First Draft of a Report on the EDVAC" (1945) escrito por el conocido matemático John von Neumann en el que se proponía el concepto de programa almacenado. Dicho documento fue redactado en vistas a la construcción del sucesor de la computadora ENIAC y su contenido fue desarrollado por John Presper Eckert, John William Mauchly, Arthur Burks y otros, durante varios meses antes de que von Neumann redactara el borrador del informe. Es por ello que otros tecnólogos como David A. Patterson y John L. Hennessy promueven la sustitución de este término por el de arquitectura Eckert-Mauchly.

3.1.1.3. Desarrollo del Concepto de Programa Almacenado

El matemático Alan Turing, quien había sido alertado de un problema de lógica matemática por las lecciones de Max Newman en la Universidad de Cambridge, escribió un artículo en 1936 titulado "On Computable Numbers, with an Application to the Entscheidungs problem"[1], que fue publicado en los "Proceedings of the London Mathematical Society". En él describía una máquina hipotética que llamó "máquina computadora universal", y que ahora es conocida como la "Máquina de Turing". La máquina hipotética tenía un almacenamiento infinito (memoria en la terminología actual) que contenía tanto las instrucciones como los datos. El ingeniero alemán Konrad Zuse escribió de forma independiente sobre este concepto en 1936. von Neumann conoció a Turing cuando ejercía de profesor sustituto en Cambridge en 1935 y también durante el año que Turing pasó en la Universidad de Princeton en 1936-37. No está claro cuándo es que él supo del artículo de 1936 de Turing.

[1] http://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

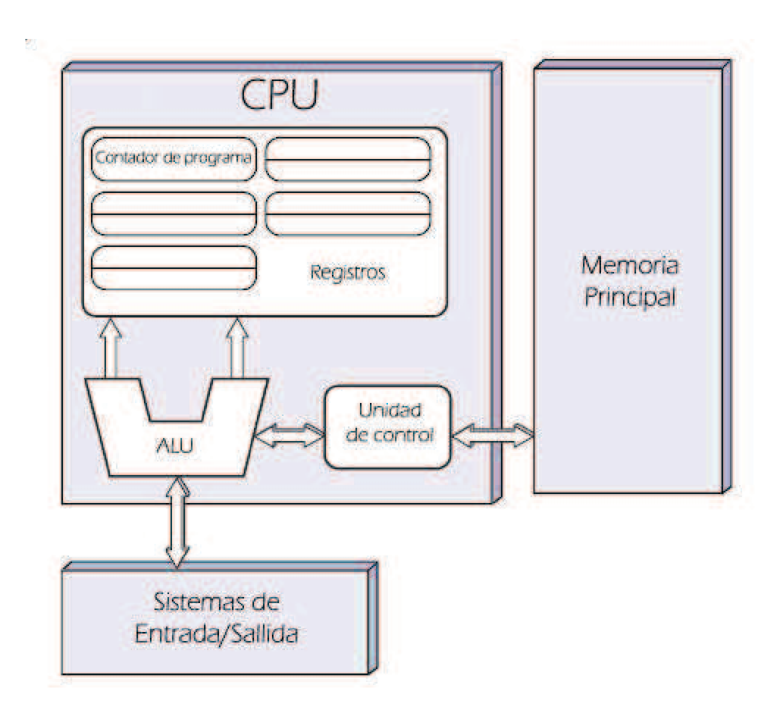


Figura 3.3: Arquitectura de von Neumann

Independientemente, J. Presper Eckert y John Mauchly, quienes estaban desarrollando el ENIAC en la Escuela Moore de Ingeniería Eléctrica en la Universidad de Pennsylvania, escribieron sobre el concepto de "programa almacenado" en diciembre de 1943. Mientras diseñaba una nueva máquina, EDVAC, Eckert escribió en enero de 1944 que se almacenarían datos y programas en un nuevo dispositivo de memoria direccionable, una línea de retardo de mercurio. Esta fue la primera vez que se propuso la construcción de un programa almacenado práctico. Por esas fechas, no tenían conocimiento del trabajo de Turing.

von Neumann estaba involucrado en el Proyecto Manhattan en el Laboratorio Nacional Los Álamos, el cual requería ingentes cantidades de cálculos. Esto le condujo al proyecto ENIAC, en verano de 1944. Allí se incorporó a los debates sobre el diseño de un computador con programas almacenados, el EDVAC. Como parte del grupo, se ofreció voluntario a escribir una descripción de él. El término "von Neumann architecture" surgió del primer artículo de von Neumann: "First

Draft of a Report on the EDVAC", fechado el 30 de junio de 1945, el cual incluía ideas de Eckert y Mauchly[2]. Estaba inconcluso cuando su compañero Herman Goldstine lo hizo circular con solo el nombre de von Neumann en él, para consternación de Eckert y Mauchly. El artículo fue leído por docenas de compañeros de trabajo de von Neumann en América y Europa, e influenció la siguiente hornada de diseños de computadoras.

Posteriormente, Turing desarrolló un informe técnico detallado, "Proposed Electronic Calculator", describiendo el Motor de Computación Automático (Automatic Computing Engine, ACE). Presentó éste al Laboratorio Nacional de Física Británico el 19 de febrero de 1946. A pesar de que Turing sabía por su experiencia de guerra en el Parque Bletchley que su propuesta era factible, el secretismo mantenido durante muchas décadas acerca de los computadores Colossus le impidió manifestarlo. Varias implementaciones exitosas del diseño ACE fueron producidas. Los trabajos de ambos, von Neumann y Turing, describían computadores de programas almacenados, pero al ser anterior el artículo de von Neumann, consiguió mayor circulación y repercusión, así que la arquitectura de computadoras que esbozó adquirió el nombre de "arquitectura von Neumann".

3.1.1.4. Descripción del Concepto de Programa Almacenado

Los primeros computadores constaban de programas almacenados. Algunos muy simples siguen utilizando este diseño; por ejemplo, una calculadora es un computador que tiene un programa almacenado. Puede hacer operaciones matemáticas simples, pero no puede ser usada como procesador de textos o videoconsola.

Cambiar el programa que contenían los dispositivos que usaban esta tecnología requería rescribir, restructurar y/o rediseñar el dispositivo. Los primeros computadores no estaban lo suficiente programados cuando fueron diseñados. La tarea de reprogramar, cuando era posible, era un proceso laborioso, empezando con notas en papel y siguiendo con detallados diseños de ingeniería. Y tras esto llegaba el, a veces, complicado proceso de rescritura y restructuramiento físico del computador.

El concepto de programa almacenado cambió por completo, se pensó en un computador que en su diseño contenía un conjunto de instrucciones que podían ser almacenadas en memoria, o sea, un programa que detallaba la computación del mismo.

[2] http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf

El diseño de un programa almacenado también daba la posibilidad a los programas de ser modificados ellos mismos durante su ejecución. Uno de los primeros motivos para su creación fue la necesidad de un programa que incrementara o modificara las direcciones de memoria de algunas instrucciones, las cuales tenían que ser hechas manualmente en los primeros diseños.

Esto se volvió menos importante cuando el índice de registros y el direccionamiento indirecto se convirtieron en algo habitual en la arquitectura de computadores. El código automodificable fue en gran parte ganando posiciones.

A gran escala, la habilidad de tratar instrucciones como datos es lo que hacen los ensambladores, compiladores y otras herramientas de programación automáticas. Se pueden "escribir programas para escribir programas".

Existen inconvenientes en el diseño de von Neumann. Las modificaciones en los programas podía ser algo perjudicial, por accidente o por diseño. En algunos simples diseños de computador con programas almacenados, un mal funcionamiento del programa puede dañar el computador. Otros programas, o el sistema operativo, posiblemente puedan llevar a un daño total en el computador. La protección de la memoria y otras formas de control de acceso pueden ayudar a proteger en contra de modificaciones accidentales y/o maliciosas de programas.

3.1.2. Firmware

El firmware es un bloque de instrucciones de máquina para propósitos específicos, grabado en una memoria de tipo de solo lectura (ROM, EEPROM, flash, etc), que establece la lógica de más bajo nivel que controla los circuitos electrónicos de un dispositivo de cualquier tipo. Está fuertemente integrado con la electrónica del dispositivo siendo el software que tiene directa interacción con el hardware: es el encargado de controlarlo para ejecutar correctamente las instrucciones externas. El programa BIOS de una computadora es un firmware cuyo propósito es activar una máquina desde su encendido y preparar el entorno para cargar un sistema operativo en la memoria RAM.

El término «firmware» fue acuñado por Ascher Opler en un artículo de 1967 publicado en Datamation. Originalmente, se refería al microcódigo - contenido en un almacenamiento de control escribible (una área pequeña especializada de memoria RAM), que definía e implementaba el conjunto de instrucciones del computador. Si fuera necesario, el firmware podía ser recargado para especializar o para modificar las instrucciones que podría ejecutar la Unidad Central

de Procesamiento (CPU). Según el uso original, el firmware contrastaba tanto con el soporte físico (la CPU en sí misma) como con el software (las instrucciones normales que se ejecutan en una CPU). El firmware no estaba compuesto de instrucciones de máquina de la CPU, sino del microcódigo de nivel inferior implicado en la implementación de las instrucciones de máquina que iría a ejecutar la CPU. El firmware existía en el límite o frontera entre el hardware y el software, por ello el término de firmware (que significa "software firme, fijo, o sólido"). Posteriormente, el término fue ensanchado para incluir cualquier tipo de microcódigo, ya fuera en RAM o ROM. Aún más adelante, el término fue ensanchado otra vez más, en el uso popular, para denotar cualquier cosa residente en ROM, incluyendo las instrucciones de máquina del procesador para el BIOS, los cargadores de arranque, o aplicaciones especializadas.



Figura 3.4: Chip de BIOS, el firmware de la mayoría de computadoras personales

El firmware ha evolucionado para significar casi cualquier contenido programable de un dispositivo de hardware, no sólo código de máquina para un procesador, sino también configuraciones y datos para los circuitos integrados para aplicaciones específicas (ASIC), dispositivos de lógica programable, etc. Hasta mediados de la década del 90, el procedimiento típico para actualizar un firmware a una nueva versión era remplazar el medio de almacenamiento que contenía el firmware, usualmente un chip de memoria ROM enchufado en un socket. Hoy en día este procedimiento no es habitual ya que los fabricantes han añadido una nueva funcionalidad que permite grabar las nuevas instrucciones en la misma memoria, haciendo de la actualización un proceso mucho más cómodo y dinámico. Aun así el proceso de actualización de un firmware hay que realizarlo con mucho cuidado, ya que al ser un componente vital cualquier fallo puede dejar al equipo inservible. Por ejemplo, un fallo de alimentación a mitad del proceso de

actualización evitaría la carga completa del código que gobierna el equipo, quizá incluso la carga del código que se encarga de actualizar el firmware, así que no podríamos actualizarlo de nuevo y por lo tanto el equipo dejaría de funcionar.

3.1.3. Lenguaje Ensamblador

El lenguaje ensamblador, o assembler (assembly language en inglés) es un lenguaje de programación de bajo nivel para los computadores, microprocesadores, microcontroladores, y otros circuitos integrados programables. Implementa una representación simbólica de los códigos de máquina binarios y otras constantes necesarias para programar una arquitectura dada de CPU y constituye la representación más directa del código de máquina específico para cada arquitectura legible por un programador. Esta representación es usualmente definida por el fabricante de hardware, y está basada en los mnemónicos que simbolizan los pasos de procesamiento (las instrucciones), los registros del procesador, las posiciones de memoria y otras características del lenguaje. Un lenguaje ensamblador es por lo tanto específico a cierta arquitectura de computador física (o virtual). Esto está en contraste con la mayoría de los lenguajes de programación de alto nivel, que, idealmente son portables.

Un programa utilitario llamado ensamblador es usado para traducir sentencias del lenguaje ensamblador al código de máquina del computador objetivo. El ensamblador realiza una traducción más o menos isomorfa (un mapeo de uno a uno) desde las sentencias mnemónicas a las instrucciones y datos de máquina. Esto está en contraste con los lenguajes de alto nivel, en los cuales una sola declaración generalmente da lugar a muchas instrucciones de máquina. Muchos sofisticados ensambladores ofrecen mecanismos adicionales para facilitar el desarrollo del programa, controlar el proceso de ensamblaje y la ayuda de depuración. Particularmente, la mayoría de los ensambladores modernos incluyen una facilidad de macro (descrita más abajo), y son llamados macro ensambladores. Fue usado principalmente en los inicios del desarrollo de software, cuando aún no se contaba con potentes lenguajes de alto nivel y los recursos eran limitados.

Actualmente se utiliza con frecuencia en ambientes académicos y de investigación, especialmente cuando se requiere la manipulación directa de hardware, altos rendimientos, o un uso de recursos controlado y reducido. Muchos dispositivos programables (como los microcontroladores) aún cuentan con el ensamblador como la única manera de ser manipulados.



Figura 3.5: Ejemplo de lenguaje ensamblador

3.2. Arquitectura de Software

3.2.1. Tiempos de Unión

Las uniones pueden ocurrir en distintos momentos, desde el punto de definición del lenguaje al tiempo de la ejecución del programa. El momento en que el enlace se produce se denomina el tiempo de unión. Hay cuatro tiempos de unión distintos:

3.2.1.1. Tiempo de Diseño del Lenguaje

Gran parte de la estructura de un lenguaje de programación se fija en el tiempo de diseño. Los tipos de datos, las estructuras de datos, comandos y formas de expresión y la estructura del programa son ejemplos de las características del lenguaje que se fijan en el momento de diseño del lenguaje. La mayoría de los lenguajes de programación dan la posibilidad de ampliar el lenguaje mediante el establecimiento de tipos de datos, expresiones y comandos definidos por el programador.

3.2.1.2. Tiempo de Implementación del Lenguaje

Algunas características del lenguaje son determinadas por la implementación del mismo. Los programas que se ejecutan en un equipo pueden no correr o dar resultados incorrectos cuando se ejecutan en otra máquina. Esto ocurre cuando el hardware difiere en su representación de los números y la aritmética. Por ejemplo, el MAXINT de Pascal está determinado por la implementación. El lenguaje de programación C proporciona acceso a la máquina subyacente y, por lo tanto, los programas que dependen de las características de la máquina subyacente pueden no funcionar como se espera cuando se trasladan a otra máquina.

3.2.1.3. Tiempo de Traducción del Programa

La unión entre el código fuente y el código objeto se produce en el momento de la traducción del programa. Las variables definidas por el programador y los tipos son un ejemplo de las uniones que se producen en el tiempo de la traducción del programa.

3.2.1.4. Tiempo de Ejecución del Programa

La unión de valores a las variables y los parámetros formales a los parámetros reales se producen durante la ejecución del programa. La unión temprana con frecuencia permite la ejecución más eficiente de los programas, mientras que la unión tardía permite más flexibilidad. La implementación de la recursión puede requerir la asignación de memoria en tiempo de ejecución a diferencia de asignación de memoria en tiempo de compilación.

3.2.2. Máquinas Virtuales

Una máquina virtual (VM) es una implementación en software de una máquina (es decir, un computador) que ejecuta los programas como una máquina física. Las máquinas virtuales se dividen en dos grandes categorías, en función de su uso y grado de correspondencia a cualquier máquina real. Una máquina virtual proporciona una plataforma completa del sistema que apoya la ejecución de un sistema operativo completo (SO). En contraste, una máquina de proceso virtual está diseñada para ejecutar un solo programa, lo que significa que soporta

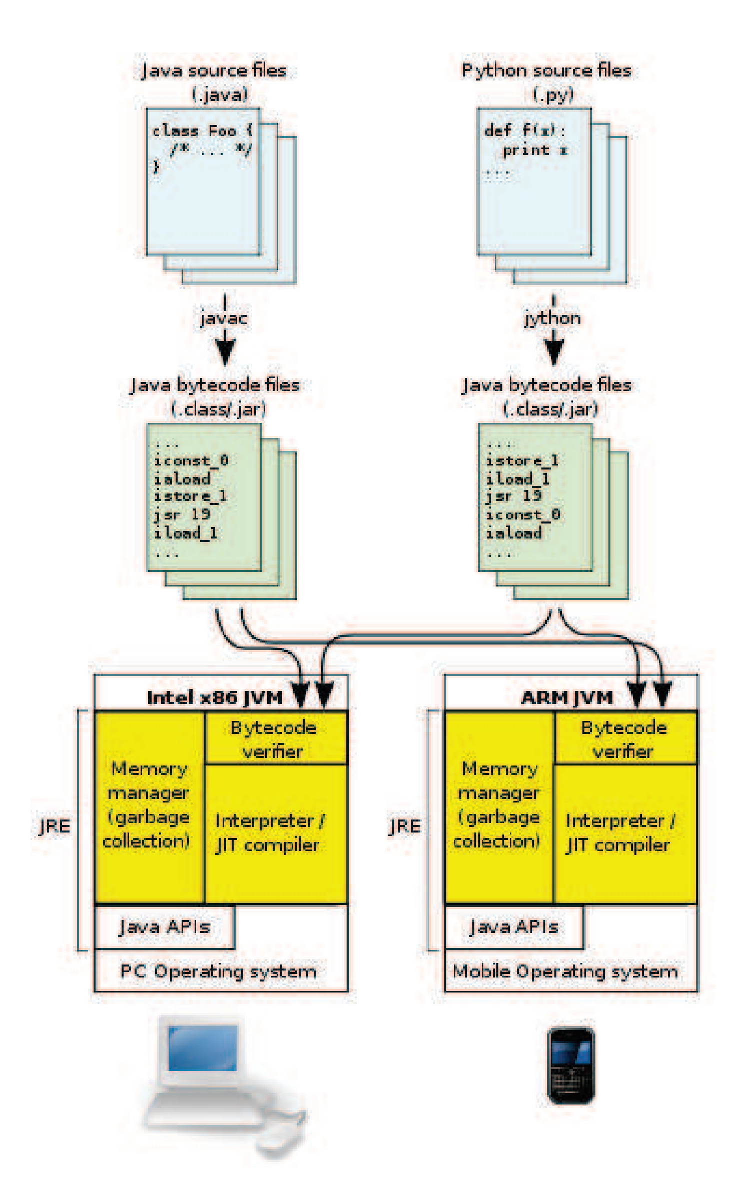


Figura 3.6: Arquitectura de la Máquina Virtual de Java

un solo proceso. Una característica esencial de una máquina virtual es que el software que se ejecuta en el interior se limita a los recursos y abstracciones proporcionados por la máquina virtual y no puede salir de su mundo virtual. Una máquina virtual se definió originalmente por Popek y Goldberg como "un duplicado eficiente y aislado de una máquina de verdad". Actualmente, las máquinas virtuales no necesariamente tienen correspondencia directa con ningún hardware real.

3.2.2.1. Máquina Virtual de Java

Una Máquina Virtual de Java (en inglés Java Virtual Machine, JVM) es un máquina virtual de proceso nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (el bytecode Java), el cual es generado por el compilador del lenguaje Java

El código binario de Java no es un lenguaje de alto nivel, sino un verdadero código máquina de bajo nivel, viable incluso como lenguaje de entrada para un microprocesador físico. Como todas las piezas del rompecabezas Java, fue desarrollado originalmente por Sun Microsystems.

La JVM es una de las piezas fundamentales de la plataforma Java. Básicamente se sitúa en un nivel superior al Hardware del sistema sobre el que se pretende ejecutar la aplicación, y éste actúa como un puente que entiende tanto el bytecode, como el sistema sobre el que se pretende ejecutar. Así, cuando se escribe una aplicación Java, se hace pensando que será ejecutada en una máquina virtual Java en concreto, siendo ésta la que en última instancia convierte de código bytecode a código nativo del dispositivo final.

La gran ventaja de la máquina virtual de Java es aportar portabilidad al lenguaje. Desde Sun Microsystems se han creado diferentes máquinas virtuales Java para diferentes arquitecturas y así un programa ."class" escrito en Windows puede ser interpretado en un entorno Linux. Tan solo es necesario disponer de una máquina virtual para dichos entornos. De ahí el famoso axioma que sigue a Java, "escríbelo una vez, ejecútalo en cualquier parte", o "Write once, run anywhere".

Pero, los intentos de la compañía propietaria de Java y productos derivados de construir microprocesadores, que aceptaran el Java bytecode como su lenguaje de máquina, fueron más bien infructuosos.

La máquina virtual de Java puede estar implementada en software, hardware, una herramienta de desarrollo o un Web browser; lee y ejecuta código precompilado bytecode que es independiente de la plataforma multiplataforma. La JVM provee definiciones para un conjunto de instrucciones, un conjunto de registros, un formato para archivos de clases, la pila, un heap con recolector de basura y un área de memoria. Cualquier implementación de la JVM que sea aprobada por Sun debe ser capaz de ejecutar cualquier clase que cumpla con la especificación.

Existen varias versiones, en orden cronológico, de la máquina virtual de Java. En general la definición del Java bytecode no cambia significativamente entre versiones, y si lo hacen, los desarrolladores del lenguaje procuran que exista compatibilidad hacia atrás con los productos anteriores.

Capítulo 4

Gramáticas Formales

La herramienta más poderosa que se tiene para la comunicación es el lenguaje. Esto es verdad si se considera la comunicación entre dos humanos, entre un programador humano y una computadora, o entre una red de computadoras. En computación se utiliza el lenguaje para describir procedimientos y se usan máquinas para pasar de descripciones de procedimientos a ejecución de procesos.

Un lenguaje es un conjunto de formas y significados, y un mapeo entre la forma y sus significados asociados. En los lenguajes naturales iniciales, las formas eran sonidos pero las formas pueden ser cualquier cosa que pueda ser percibida por las partes que se comunican tales como golpes de tambor, gestos manuales o figuras.

Un lenguaje natural es un lenguaje hablado por humanos, tal como español o swahili. Los lenguajes naturales son muy complejos debido a que han evolucionado por muchos miles de años de interacción cultural e individual. El enfoque en este libro será en lenguajes diseñados y creados por humanos con el propósito específico de expresar procedimientos a ser ejecutados por una computadora. El enfoque será en lenguajes donde las formas son el texto. En un lenguaje textual, las formas son secuencias lineales de caracteres. Una cadena es una secuencia de cero o más caracteres. Cada caracter es un símbolo extraído de un conjunto finito denominado alfabeto. Para español, el alfabeto es el conjunto {a, b, c, . . . , z} (para el lenguaje completo, mayúsculas, numerales, y símbolos de puntación son también necesarios).

Forma	Significado
Verde	Pasar
Amarillo	Precaución
Rojo	Parar

Cuadro 4.1: Forma y significado de un semáforo

Un sistema de comunicación sencillo puede ser descrito usando una tabla de formas y sus significados asociados. Por ejemplo, el cuadro 4.1 describe un sistema de comunicación entre semáforos y choferes:

Los sistemas de comunicaciones que involucran a humanos son notoriamente imprecisos y subjetivos. Un chofer y un oficial de policía pueden estar en desacuerdo en el significado actual del símbolo amarillo y pueden aún estar en desacuerdo en cuál símbolo es transmitido por el semáforo en un momento particular. Los sistemas de comunicaciones para computadoras necesitan precisión; si se quiere conocer lo que el programa ejecutará, es importante que cada paso que se ejecute sea entendido de manera precisa y sin ambigüedades.

El método de definir un sistema de comunicación por una Tabla de Pares

$$<\text{Símbolo, Significado}>$$

puede trabajar adecuadamente solamente para sistemas de comunicaciones triviales. El número de posibles significados que pueden ser expresados está limitado por el número de entradas en la Tabla. Es imposible expresar cualquier nuevo significado debido a que todos los significados deben ya estar listados en la Tabla.

Un lenguaje útil debe ser capaz de expresar de infinitas maneras muchos significados diferentes; entonces, debe haber una manera de generar nuevas formas con sus significados. No hay representación finita, tal como en una tabla impresa que pueda contener todas las formas y significados de un lenguaje infinito. Una manera de generar un número infinito de conjuntos largos es utilizar repetidamente patrones. Por ejemplo: la mayor parte de los humanos puede interpretar la notación "1, 2, 3,..." como el conjunto de todos los números naturales. Se puede interpretar "..." como un significado que quiere decir que continúa siendo lo mismo por siempre. En este caso, significa mantener añadiendo uno al número anterior. Entonces, solamente con unos pocos números y símbolos se puede escribir un conjunto que contiene de manera infinita muchos números.

El lenguaje de los números naturales es suficiente para codificar todos los significados de cualquier conjunto contable, pero encontrar un mapeo sensible entre la mayor parte de los significados y los números es casi imposible. Las formas no corresponden suficientemente cerca a las ideas que se quieren expresar para tener un leguaje útil.

4.1. Construcción de un Lenguaje

Para definir de una manera más expresiva lenguajes infinitos, se requiere de un sistema más rico para construir nuevas formas y sus significados asociados. Se necesitan maneras de describir lenguajes que permitan definir un conjunto infinito de formas y significados con una notación compacta. La metodología que se utiliza es especificar un lenguaje definiendo un conjunto de reglas que producen exactamente el mismo conjunto de formas en el lenguaje.

4.1.1. Componentes de un Lenguaje.

Un lenguaje está compuesto de:

- Primitivos: La unidad más pequeña de significado.
- Significados de combinaciones: Reglas para construir nuevos elementos del lenguaje al combinar los más sencillos.

Los Primitivos son las unidades significativas más pequeñas. Un Primitivo no puede ser dividido en partes más pequeñas cuyos significados puedan ser combinados para producir el mismo significado de la unidad. Esto significa que las combinaciones son reglas para construir palabras con primitivos y para construir frases y sentencias desde las palabras.

Debido a que se tiene reglas para producir nuevas palabras, no todas las palabras son primitivas. Por ejemplo: se puede crear una nueva palabra añadiéndole anti al frente de una palabra existente.

El significado de la nueva palabra puede ser inferido como "contra el significado de la palabra original". Reglas como ésta significan que cualquiera puede

inventar una nueva palabra y usarla en la comunicación, de manera que probablemente será entendida por los que escuchan y nunca anteriormente han escuchado la palabra.

Por ejemplo, la palabra congelar significa pasar del estado liquido al estado sólido. Anticongelar es la acción diseñada para prevenir congelamiento. Las personas que conocen el significado de congelar y anti pueden, de manera general, entender el significado de anticongelar, aun cuando ellos nunca hayan escuchado la palabra anterior.

Primitivos son las unidades más pequeñas de significado no basado en la forma; así como anti y congelar son primitivos, ellos no pueden ser divididos en partes más pequeñas con significado. Se puede dividir anti en 2 sílabas o 4 letras, pero estos subcomponentes no tienen significado que permita combinar o producir el significado del primitivo.

4.1.2. Medios de Abstracción

Adicionalmente a los primitivos y medios de combinación, los lenguajes más poderosos tienen un tipo adicional de componente que les permiten una económica comunicación: medios de abstracción.

Los medios de abstracción permiten dar un nombre simple a una entidad compleja. En español, los significados de abstracción, por ejemplo, son pronombres como ella, él, ellos. El significado de un pronombre depende del contexto en el cual es utilizado. Resume el significado complejo en una sola palabra. Por ejemplo, "él" en una sentencia significa "el significado del pronombre", "él" en otra sentencia significa "el pronombre".

En los lenguajes naturales existe un número limitado de medios de abstracción. El español tiene un conjunto muy limitado de pronombres para abstraer personas. Tiene: ella y él, para abstraer una persona femenina o masculina, respectivamente, pero no existen pronombres neutrales para abstraer una persona de cualquier sexo. La interpretación de lo que es una abstracción de pronombre en lenguaje natural es a menudo confusa. Por ejemplo: no es claro a que "él" en esta oración se refiere. Lenguajes para programación de computadoras necesitan que la abstracción deba ser poderosa y no-ambigua.

4.2. Etapas de Traducción de un Lenguaje

Todo software que se ejecuta en una computadora ha sido escrito en un lenguaje de programación de bajo o alto nivel. Para que este programa pueda ser ejecutado es necesario traducirlo a un código binario que pueda ser ejecutado por el computador. El proceso de convertir el programa que se encuentra en formato de texto (programa fuente) a un lenguaje intermedio o a lenguaje de máquina se lo denomina Traducción.

Existen diferentes formas de traductores de lenguajes; de manera general se puede decir que el proceso de traducción es el de la figura 4.1.

El preprocesador se utiliza en algunos lenguajes que son previamente modificados al entrar al proceso de compilación; por ejemplo, un programa escrito en C++ es la entrada al preprocesador y la salida es un programa en lenguaje C que es la entrada a un compilador C.

El proceso de compilación, descrito más adelante, genera como salida un programa en lenguaje ensamblador que posteriormente, por medio del ensamblador genera el programa en lenguaje de máquina. Este proceso es dependiente del procesador en el que se va a ejecutar el programa. Lenguajes como Pascal, Smalltalk, Java o Ruby independizan el procesamiento de la máquina compilando a un lenguaje intermedio que es interpretado en una máquina virtual. La máquina virtual es sencilla de implementar y es dependiente del procesador.

El programa, en lenguaje de máquina, es enlazado con archivos de biblioteca y con otros módulos para obtener el programa ejecutable.

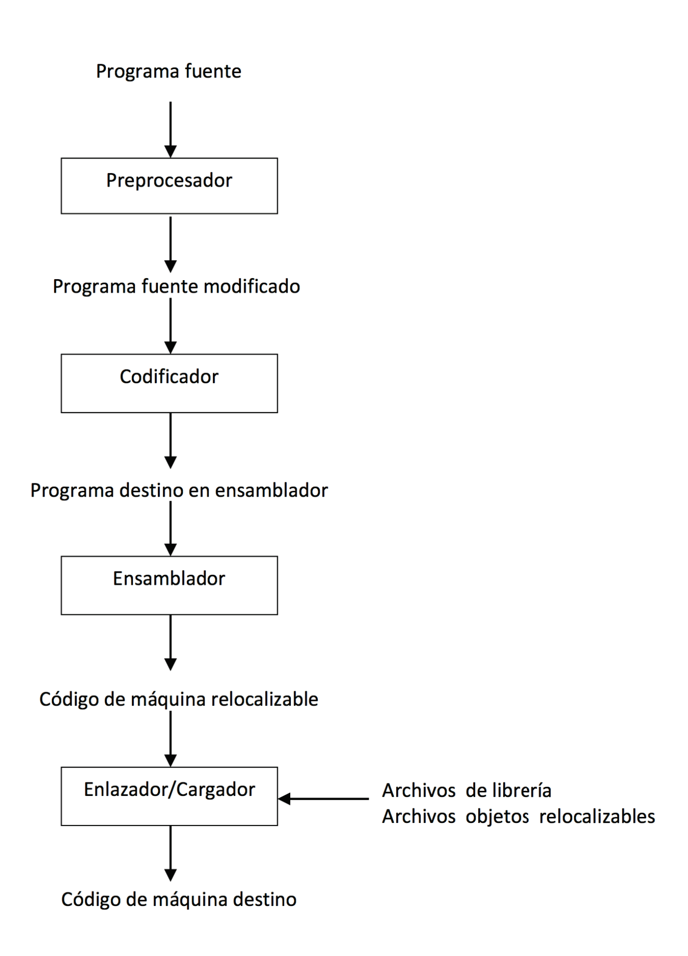


Figura 4.1: Sistema de procesamiento de lenguaje

4.2.1. La Estructura del Compilador

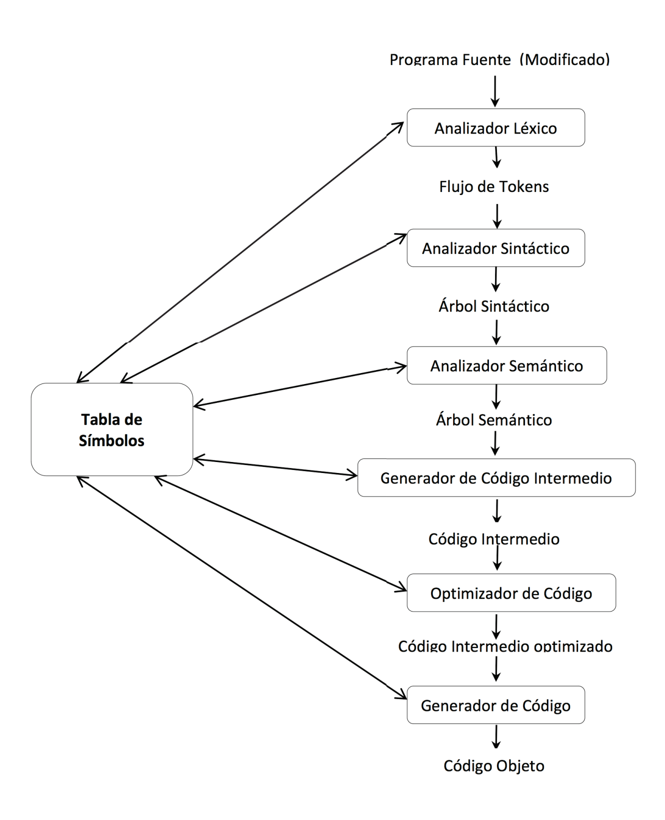


Figura 4.2: Fases de un compilador

La parte más importante del procesamiento del programa es la traducción del programa fuente a programa objeto mediante un compilador.

El compilador tiene una parte de análisis que procesa el programa fuente y revisa su estructura gramatical. Si en la fase de análisis se detectan errores en su parte sintáctica o semántica, el compilador debe generar la información adecuada para que pueda ser corregido. En esta fase, la información recolectada

queda almacenada en la tabla de símbolos que posteriormente es utilizada en la fase de síntesis.

La síntesis, a partir de la representación intermedia y de la tabla de símbolos, genera el programa objeto.

El proceso de optimización es opcional; para ahorrar tiempo, algunos compiladores tienen un mecanismo para desactivar este proceso, mientras el programa da errores de compilación y se activa la optimización una vez que el programa ha sido compilado con éxito.

4.2.1.1. Análisis Léxico.

La primera fase del compilador es el análisis léxico. El analizador barre (escanea) el programa fuente leyendo cada línea de texto como un flujo de caracteres, que los agrupa en lexemas; para cada lexema el analizador léxico produce un token de la forma:

<nombre-token, valor-atributo>

La información del token queda almacenada en la tabla de símbolos en la dirección valor-atributo y es actualizada en las siguientes fases del análisis. Por ejemplo, en la siguiente asignación:

a = b + 2 * c

Al barrer el texto se encuentran los siguientes lexemas: a, =, b, +, 2, *, c; generándose los siguientes tokens:

a <id, 1>

= <=>

b <id, 2>

+ <+>

2 <num, 3>

* <*>

C <id, 4>

Donde el símbolo id indica el tipo de token identificador y num es un token tipo número

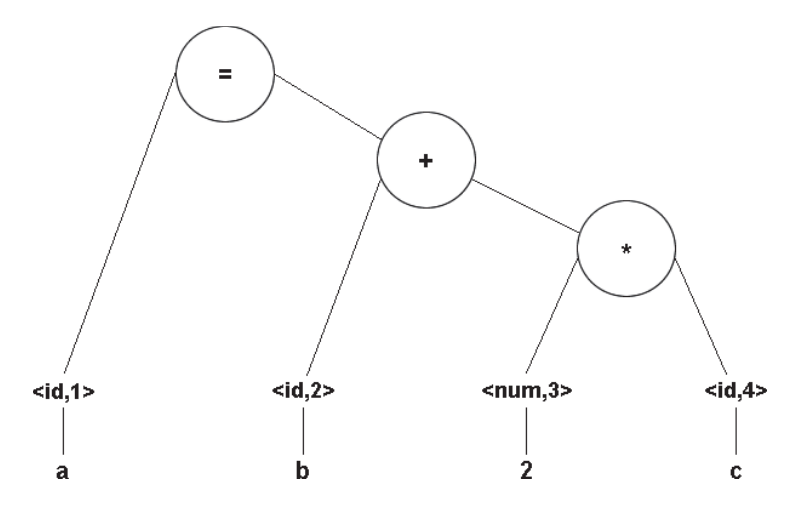


Figura 4.3: Árbol sintáctico

4.2.1.2. Análisis Sintáctico

El analizador sintáctico (parser) en base al flujo de tokens y a la tabla de símbolos genera el árbol sintáctico que describe la estructura gramatical y la precedencia de las operaciones a ser ejecutadas, el árbol sintáctico tiene como hojas los identificadores y números y como nodos los operadores. La gramática del lenguaje establece que la precedencia de los operadores es de la multiplicación (*), suma (+) y asignación (=)

4.2.1.3. Análisis Semántico

El análisis sintáctico comprueba que el programa cumpla las reglas gramaticales del lenguaje. El análisis semántico utiliza el árbol sintáctico y la tabla de símbolos para chequear la semántica del programa; por ejemplo, el chequeo de tipos. El árbol sintáctico de la Figura 4.3 no comprueba los tipos de las operaciones; "c" puede ser un número real por lo que la operación 2*c puede generar un error semántico.

4.2.1.4. Generación de Código Intermedio

La generación de código intermedio es la primera parte del proceso de síntesis. La generación de código intermedio puede ser de nivel bajo (lenguaje ensamblador) dependiente del lenguaje de máquina del procesador en el que se va a ejecutar el programa. La generación del código puede ser de nivel intermedio, independiente del procesador, a ser ejecutado en una máquina virtual. Una versión de Pascal compila a lenguaje intermedio P, que es interpretado en una máquina virtual de pilas; Java es compilado a bytecode y ejecutado en la máquina virtual JVM (Java Virtual Machine). La velocidad de ejecución en una máquina virtual es menor comparada con la ejecución de un programa en lenguaje de máquina, pero la máquina virtual permite la portabilidad y seguridad del programa para que pueda ser ejecutado en diversos tipos de sistemas operativos y de procesadores.

4.2.1.5. Optimización de Código

La optimización de código, proceso opcional, trata de mejorar el uso de memoria o la velocidad de ejecución del programa. Si tenemos, en una malla, una asignación a una variable de un valor constante que no es modificado durante la ejecución de la malla, la optimización consiste en sacar esa asignación fuera de la malla para que sea ejecutada solo una vez.

4.2.1.6. Generación de Código

El generador de código tiene como salida el programa objeto que puede ser en lenguaje de máquina dependiente del juego de instrucciones del procesador. Este proceso implica un manejo adecuado de la memoria y de los registros del procesador de tal forma que el almacenamiento de las variables para la ejecución de las operaciones sea adecuado.

4.2.1.7. Tabla de Símbolos

La tabla de símbolos es utilizada a lo largo de todo el proceso por lo que la estructura de datos que se utilice para almacenar, modificar y recuperar la información para cada nombre de variable debe ser eficiente en velocidad.

El desarrollo de un compilador es el desarrollo de un sistema de software por lo que se pueden aplicar herramientas computacionales para, en ciertos casos, automatizar este proceso. El proceso de análisis léxico y sintáctico para gramáticas regulares se puede desarrollar con un Generador de Analizadores Sintácticos; el más conocido es yacc (yet another compiler compiler) que es un compilador de compiladores. Esta herramienta es muy utilizada en C para desarrollar compiladores.

4.3. Tipos de Gramática

La gramática formal define, mediante reglas de formación, cómo estructurar las oraciones de un lenguaje formal. Estas reglas describen cómo estructurar las oraciones a partir del alfabeto del lenguaje; estas oraciones son válidas en base a la sintaxis del lenguaje. La gramática no describe el significado de la oración, solamente su forma. Las siguientes oraciones:

Pedro trabajó la tierra

La tierra trabajó la casa

son sintácticamente correctas; sin embargo, la segunda oración semánticamente no lo es.

El lingüista Noam Chomsky en la década de los 50 introdujo la teoría de los lenguajes formales,[1] en la que desarrolló herramientas para estudiar y formalizar los lenguajes naturales. Chomsky clasificó los lenguajes en una jerarquía de cuatro tipos diferentes de gramática.

- Gramáticas de tipo cero. Estas gramáticas, que son las más generales, no tienen restricción alguna e incluyen todas las gramáticas formales. Los lenguajes generados pueden ser reconocidos por la máquina de Turing.
- Gramáticas de tipo uno. Corresponden a lenguajes sensibles al contexto; existen limitaciones en la estructura gramatical y, el valor sintáctico de la palabra depende de su posición en la oración. Este tipo de gramática es un subconjunto de la de tipo cero.

[1] Chomsky, Noam (1956). "Three models for the description of language". IRE Transactions on Information Theory (2): 113–124. doi:10.1109/TIT.1956.1056813

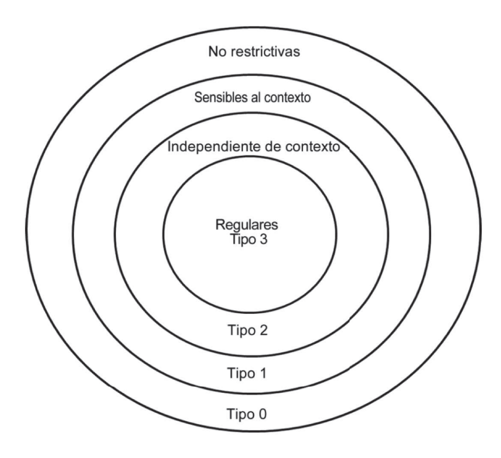


Figura 4.4: La Jerarquía de Chomsky

- Gramáticas de tipo dos. Éstas corresponden a lenguajes independientes del contexto; tienen mayor limitación que las de tipo uno y el valor sintáctico de la palabra es independiente de la posición en la oración. Todos los lenguajes de programación pueden expresarse con una sintaxis correspondiente a una gramática de tipo dos. Las gramáticas de tipo dos son un subconjunto de las de tipo uno.

- Gramáticas tipo tres. Corresponden a los lenguajes regulares y expresiones regulares. Estas gramáticas pueden ser analizadas mediante un autómata finito; son las más restrictivas y son un subconjunto de las de tipo dos.

El siguiente gráfico da un resumen de los cuatro tipos de gramática:

Los lenguajes de programación son lenguajes generalmente basados en la teoría de lenguajes independientes de contexto; La gramática formal utilizada es de tipo dos o tipo tres. La mayoría de los lenguajes naturales son sensibles al contexto; sin embargo, la teoría de las gramáticas tipo uno no están todavía lo suficientemente desarrolladas como para ser aplicadas, de manera general, en los lenguajes de programación[2].

[2]http://danielmattosroberts.com/earley/context-sensitive-earley.pdf

4.4. Gramáticas Independientes (Libres) de Contexto

El desarrollo de la teoría de los lenguajes de programación se dio en forma paralela al desarrollo de la teoría de Chomsky sobre los lenguajes naturales. John Backus diseñador de FORTRAN y Peter Naur, en un reporte de 1963, desarrollaron la notación para ALGOL 60; esta notación fue llamada BNF (Backus-Naur Form).

4.4.1. Definición Formal

Las gramáticas independientes de contexto (BNF) o de tipo 2, son lo sufrientemente sencillas como para permitir la construcción eficiente de árboles sintácticos generados por el analizador léxico.

Una gramática independiente de contexto o de contexto libre G está definida por una tupla de 4:

$$G = (V, \Sigma, R, S)$$

donde:

1. V es un conjunto finito; cada elemento $v \in V$ es llamado un caracter no terminal o una variable. Cada variable define un sub-lenguaje del lenguaje definido por G.
2. Σ es el conjunto finito de terminales que constituyen los elementos de la sentencia. Este conjunto de terminales es el alfabeto (letras o símbolos) del lenguaje definido por la gramática G.
3. R es una relación finita de V a $(V \cup \Sigma)^*$ .Los miembros de R son llamados las reglas de producción de la gramática (también simbolizada por P).
4. S es el símbolo o variable inicial utilizado para representar toda la sentencia o programa. Debe ser un elemento de V.

El asterisco es la operación cerradura de Kleene (Kleene closure) que representa cero o más ocurrencias. El lenguaje de una gramática G es el conjunto L(G) = {ωε Σ* : S*=> ω}

Una regla de producción en R es un par $(\alpha, \beta) \in R$, donde $\alpha \in V$ es un no terminal y $\beta \in (V \cup \Sigma)^*$ es una cadena de variables y no terminales; las reglas de producción usualmente se las escribe como: α ::= β o $\alpha \rightarrow \beta$. β puede ser la cadena vacía que se la denomina ε o λ .

Por ejemplo la oración:

Juan compró la bicicleta

La oración tiene dos elementos: Sujeto y Predicado. Juan es el sujeto de la oración y el predicado es: compró la bicicleta. Una gramática válida para la oración estaría dada por las siguientes reglas de producción:

1. <oración> ::= <sujeto> <predicado>
2. <sujeto> ::= <sustantivo>
3. <predicado> ::= <verbo><artículo><sustantivo>
4. <sustantivo> ::= Juan | Pedro | bicicleta | gato
5. <verbo> ::= compró | montó
6. <artículo> ::= el | la

- Los símbolos no terminales están encerrados entre "<" y ">" y la operación "|" significa escoger una de las alternativas. Si a y b son dos cadenas, ab es una operación que concatena las dos cadenas
- El conjunto finito V es: {<oración>, <sujeto>, <predicado>, <sustantivo>, <verbo>, <artículo>}
- El conjunto finito de terminales o alfabeto Σ es {Juan, Pedro, bicicleta, gato, compró, montó, el, la}. Nótese que $V \bigcap \Sigma = \Phi$
- Las reglas de producción son las escritas de 1 a 6.
- El símbolo inicial es <oración>

Para encontrar si la frase pertenece a la gramática se tiene que realizar el análisis léxico y encontrar los lexemas y tokens de la frase; en esta gramática los lexemas están separados por espacios: Juan, compró, la, bicicleta. Una vez realizado el

análisis léxico se continúa con el análisis sintáctico y la generación del árbol sintáctico.

En el análisis léxico se remplaza, en el símbolo inicial, en el lado derecho, cualquier no terminal con el lado derecho de una producción que contenga ese no terminal, en el lado izquierdo. En el ejemplo, la derivación sería:

<oración>=><sustantivo><predicado>=>Juan<predicado>

=>Juan<verbo><artículo><sustantivo>

=> Juan compró<artículo><sustantivo>

=>Juan compró la <sustantivo>

=> Juan compró la bicicleta

Se ha obtenido la frase, formada por símbolos terminales, derivándola a partir del símbolo inicial <oración>. Algunas frases válidas en esta gramática:

Pedro montó la bicicleta

Juan compró el gato

La derivación fue realizada de izquierda a derecha y el árbol sintáctico generado es el de la figura 4.5.

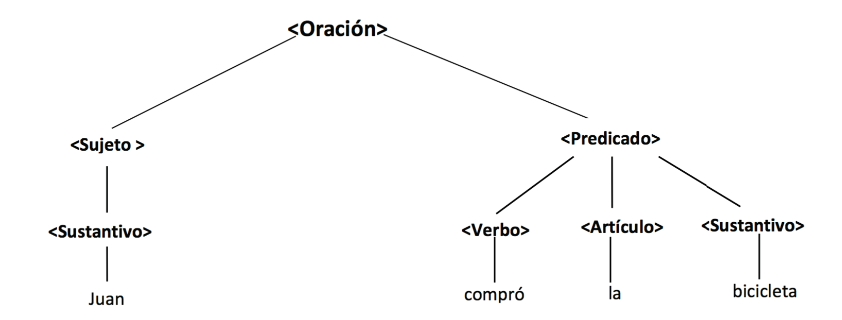


Figura 4.5: Árbol sintáctico

La siguiente producción

1. <lista> ::= <lista> + <entero>

2. <lista> ::= <lista> − <entero>
3. <lista> ::= <entero>
4. <entero> ::=< digito>|<entero><digito>
5. <digito> ::= 0|1|2|3|4|5|6|7|8|9|0

se refiere a expresiones con operaciones de suma y resta representados por símbolos terminales positivos y negativos.

Expresiones válidas: 5 + 4 − 2, 86 − 27 + 55

Se puede rescribir las reglas de producción:

1. <lista> ::= <lista> + <entero> | <lista> - <entero> | <entero>
2. <entero> ::= <digito><digito>*
3. <digito> ::= 0|1|2|3|4|5|6|7|8|9|0

El orden de precedencia de las operaciones de esta gramática es: la cerradura de Kleene, la concatenación y luego la operación |, todas ellas asociativas por la izquierda. En C la asignación = es asociativa por la derecha; es decir, a=b=c se ejecuta a = (b=c); las reglas de producción:

1. derecha → letra = derecha | letra
2. letra → a | b | c | . . . | z

La palabra cursiva indica un no terminal y → es equivalente a ::=

Una sentencia puede ser derivada de maneras diferentes pero las derivaciones están basadas en la misma estructura básica. Por ejemplo, dada la gramática:

S => AaS|A

A => b|c

la cadena

cacab

Puede ser derivada de diferentes maneras:

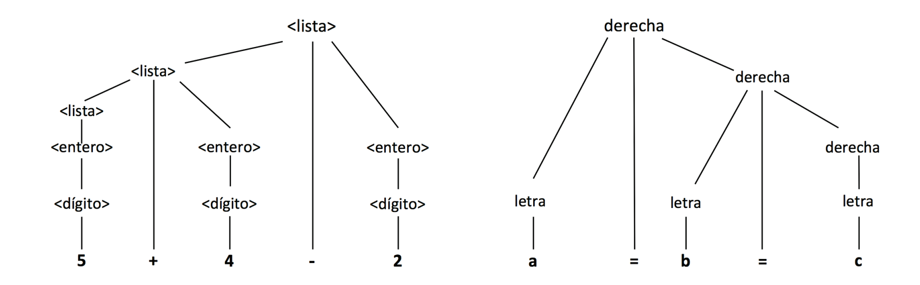


Figura 4.6: Árboles para 5 + 4 – 2 y a = b = c con gramáticas asociativas por la izquierda y por la derecha

S => AaS => AaAaS => AaAaA => caAaA => caAab => cacab

o

S => AaS => AaAaS => AaAaA => AacaA => Aacab => cacab

El árbol sintáctico para la cadena está definido por la gramática independientemente de la manera en que pueden ser derivados:

Existen dos maneras de derivar que son importantes; en la derivación por la derecha se remplaza el no terminal que está más a la derecha de cada forma. En la cadena cacab es:

S => AaS => AaAaS => AaAaA => AaAab => Aacab => cacab

Un análisis por la derecha (right parse) es el reverso de la secuencia de producciones que constituyen la derivación por la derecha. Definiciones similares son para la derivación y análisis por la izquierda.

4.4.2. Ambigüedad

Si la gramática genera más de un árbol sintáctico para una cadena se dice que la gramática es ambigua. Es suficiente que exista para una cadena dos árboles para que la gramática sea ambigua. Por ejemplo, la siguiente gramática es ambigua:

<asign> => <id>:=<expr>

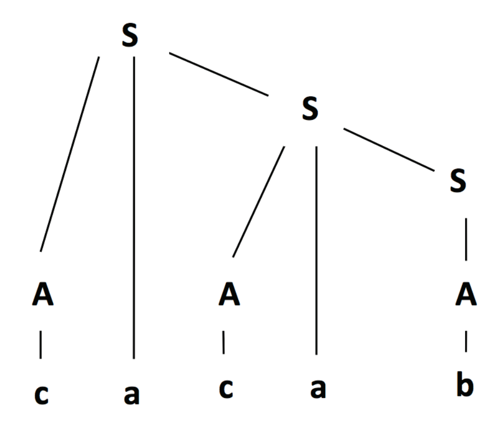


Figura 4.7: Árbol sintáctico para cacab

<id> => A|B|C

<expr> => <expr>+<term>|<term>

<term> => <term>*<factor>|<factor>

<factor> => (<expr>)|<id>

Si se construye el árbol sintáctico para A:=B+C*A se pueden construir dos árboles válidos (Figura 4.8).

Los dos árboles representan la precedencia de las operaciones; el primero B+(C*A) y el segundo (B+C)*A presentando resultados diferentes.

En el proceso de compilación, como se ha mencionado anteriormente, el analizador léxico lee el programa fuente y agrupa las secuencias de caracteres, remueve espacios irrelevantes y comentarios agrupando los caracteres en identificadores, números, palabras reservadas, etc. La salida del analizador léxico es la secuencia de tokens; un token por cada identificador, número, palabra reservada, etc.

El analizador sintáctico, para gramáticas libres de contexto, recibe como entrada una secuencia de tokens del analizador léxico y trata de determinar si la estructura del programa está acorde con la gramática del lenguaje, construyendo (de manera figurativa o literal) el árbol sintáctico.

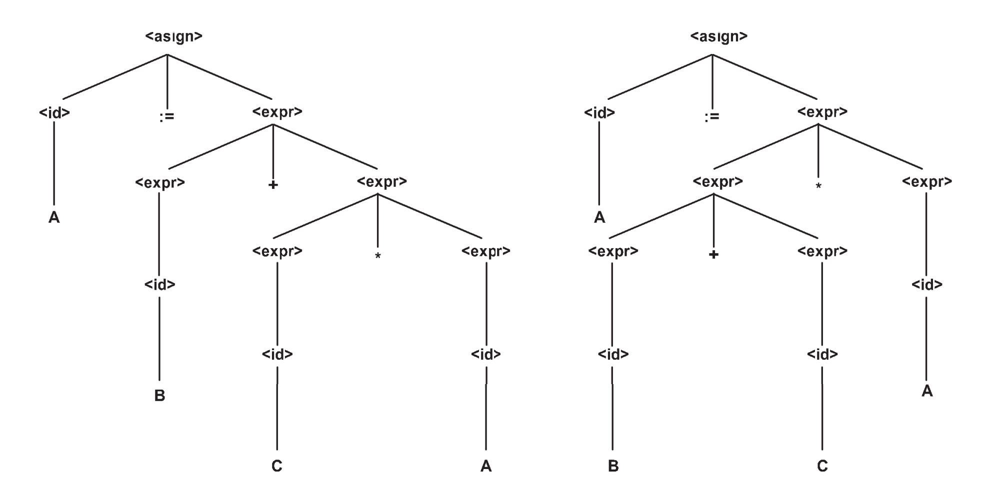


Figura 4.8: Árboles sintácticos para A:=B+C*A

La tarea de producir analizadores depende del tipo de gramática que se quiera implementar. Existen dos clases principales de gramáticas de contexto libre que se utilizan, las gramáticas LL(k) y las LR(k). La importancia de ellas es que se puede utilizar un solo barrido sobre el texto de entrada. Otra ventaja es que los analizadores pueden ser automáticamente generados (por ejemplo Yacc).

4.4.2.1. Gramáticas LL(k)

Una gramática LL(k) está definida, de manera informal, si el analizador sintáctico puede tomar una decisión basado en máximo los siguientes k símbolos de entrada; es decir se miran k símbolos adelante. Dada una producción:

$A \rightarrow \alpha_1 | \ \alpha_2 | \ldots | \ \alpha_n$

La decisión de derivar α_1 o α_2, etc. se la hace en base a los k símbolos siguientes en el flujo de entrada. El barrido del flujo es realizado de izquierda a derecha (L) y el proceso resulta en la deducción de la sintaxis izquierda de las sentencias del lenguaje (segunda L). Una gramática muy común es la LL(1). La gramática descrita por:

$A \rightarrow b|Ac$

es recursiva por la izquierda por cuánto la producción de un no terminal tiene una parte derecha que empieza con ese no terminal; esta gramática sería LL(1) si se pudiera deducir en base a un sólo caracter si se aplica la producción A→b o A→Ac, pero la letra b puede empezar cualquiera de las dos producciones, por lo que no se puede tomar una decisión al respecto con solo un símbolo. En la actualidad existen herramientas que pueden convertir algunas gramáticas no LL(1) a equivalentes LL(1).

4.4.2.2. Gramáticas LR(k)

Una gramática es LR(k) si cada sentencia que genera puede ser analizada en un solo barrido de izquierda a derecha mirando máximo k símbolos adelante. Un analizador para una gramática LR(k) tratará de construir el árbol sintáctico de abajo hacia arriba mirando por el reverso de la derivación más a la derecha (R). La L implica que el barrido es de izquierda a derecha. Dada una producción:

$A \rightarrow \alpha_1\alpha_2\ldots\alpha_n$

Tendrá que reconocer α_1 luego α_2 hasta α_n; una vez reconocidos todos, el analizador reconocerá A.

Por ejemplo, en la siguiente gramática:

1. S → A
2. A → a
3. A → Ab

La cadena abb sería analizada de la siguiente manera:

- Desplace a; reducción por A → a, producción 2
- Desplace b; reducción por A → Ab, producción 3
- Desplace b; reducción por A → Ab, producción 3
- Finalmente reducción por S → A, producción 1

Todas las gramáticas LL son también gramáticas LR pero no lo opuesto.

4.4.3. Extensiones a la Notación BNF

Si bien la notación BNF es sencilla utilizando los operadores de concatenación ab, de cerradura * y de alternativas (o) |; para expresar una gramática de manera más simple es necesario extender la notación, por lo que se han definido las siguientes notaciones adicionales:

- Opción. Un elemento opcional puede ser indicado encerrando el elemento con paréntesis rectos, [. . .].
- Alternativa. Para escoger alternativas se utiliza el símbolo | opcionalmente encerrado en paréntesis ([,])
- Repetición. Una secuencia arbitraria de instancias de un elemento puede ser representada encerrada en llave, seguida por un asterisco, {. . . }*
- Agrupación. Las agrupaciones están definidas por (. . .)

Ejemplo de uso de la notación:

<entero> ::= [+|-] <dígito>{<dígito>}*

<dígito> ::= 0|1|2|3|4|5|6|7|8|9

4.5. Autómatas Finitos

Un autómata finito es un grafo que tiene:

- Estados representados por círculos; cada estado tiene un identificador escrito dentro del círculo. Existe un número finito de estados.
- Alfabeto de entrada. Un conjunto de símbolos de entrada
- Arcos de transición de un estado a otro estado en base al alfabeto de entrada.
- Estado Inicial. Existe un solo estado inicial que es el estado donde se inicia el proceso
- Estado(s) final(es). Conjunto de estados de aceptación, están representados por un doble círculo. Los autómatas finitos pueden ser de dos tipos:

- Autómatas finitos no determinísticos. Un símbolo puede etiquetar varios arcos que salen del mismo estado, y ε es una posible etiqueta.
- Autómatas finitos determinísticos, tienen para cada estado y para cada símbolo del alfabeto un solo arco que sale de ese estado.

Por ejemplo, un autómata que reconozca, dado el alfabeto Σ={a,b}, las cadenas que no contienen dos a consecutivas.

Un autómata no determinístico:

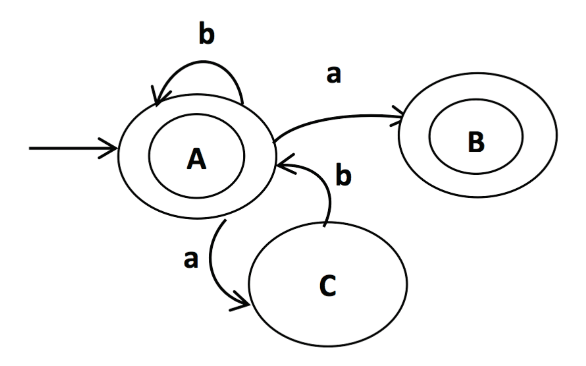


Figura 4.9: Autómata no determinístico

Un autómata determinístico:

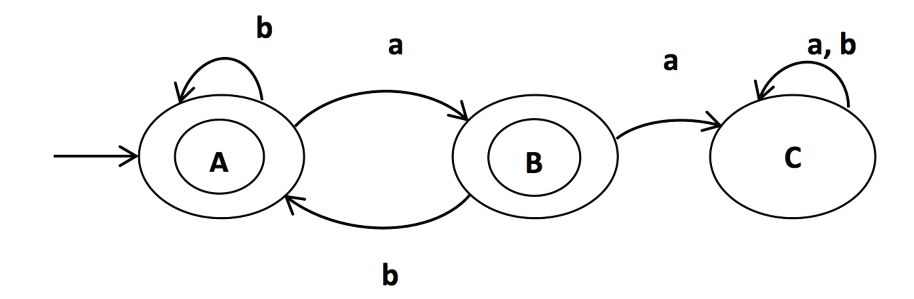


Figura 4.10: Autómata determinístico

Ambos representan el mismo lenguaje, el estado inicial es A y los estados finales son A y B; en el autómata determinístico (Figura 4.10) podemos eliminar el estado C, que es un estado de error, por cuanto una vez que se ejecuta la transición del estado B al C ya no se sale de ese estado.

Usualmente los autómatas no determinísticos tienen grafos sencillos pero son más complicados de implementar; los autómatas determinísticos tienen grafos más complicados pero más fáciles de implementar.

El siguiente autómata reconoce números binarios múltiples de tres:

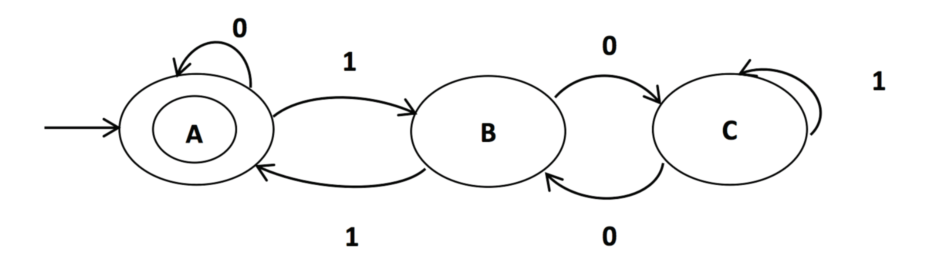


Figura 4.11: Autómata determinístico que reconoce números múltiples de tres

Los autómatas determinísticos como los no determinísticos reconocen los mismos lenguajes; aún más, estos lenguajes son los lenguajes regulares y las expresiones regulares.

4.6. Lenguajes Regulares

Los lenguajes regulares o de tipo tres, son los más restrictivos; toda gramática de tipo tres es también de tipo dos, o sea de contexto libre. Las gramáticas regulares pueden tener una de las siguientes formas:

- Gramáticas lineales por la izquierda
  - A::=a
  - A::=Ba
- Gramáticas lineales por la derecha
  - A::=a
  - A::=aB

Las producciones, en el lado derecho, están conformadas por un terminal o un terminal y un no terminal en cualquier orden. Se ha demostrado que dada una gramática lineal por la izquierda, existe otra lineal por la derecha equivalente que representa el mismo lenguaje y viceversa.

Por ejemplo, la gramática regular para generar números binarios:

A ::= 0A|1A|0|1

Las gramáticas de tipo 3 generan arboles sintácticos binarios.

Las expresiones regulares son cadenas terminales que se generan utilizando los operadores de unión, concatenación y cerradura. Cada expresión regular g define un lenguaje L(g). Las operaciones de unión, concatenación y cerradura sobre dos expresiones regulares generan una expresión regular.

La precedencia de operadores, todos asociativos por la izquierda, es:

- El operador unitario * tiene la precedencia más alta
- La concatenación tiene la siguiente precedencia
- La unión | tiene la precedencia más baja

Por ejemplo:

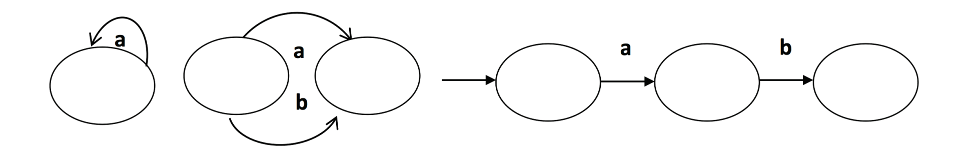


Figura 4.12: Cerradura de Kleene a*, alternabilidad a|b y concatenación ab

- a* es el conjunto de todas las cadenas de a incluyendo la cadena vacía
- (a|b)* o (a*b*)* representan las cadenas con cero o más instancias de a y b

Operadores adicionales para las expresiones regulares:

- El operador + que significa una o más veces a+=aa*
- El operador ? significa cero o una vez a? es equivalente a a|ε
- Una secuencia lógica se la define en paréntesis rectos con la abreviación [a1a2a3...an]; por ejemplo, las letras minúsculas [a-z], las minúsculas y mayúsculas [a-zA-Z]

En las gramáticas y expresiones regulares existe una relación biunívoca entre la gramática y un autómata de estados. Dada una gramática regular podemos construir un autómata, o dado un autómata podemos generar una gramática para el lenguaje.

Las tres conversiones básicas son:

Para el autómata de la Figura 4.12 la expresión regular está dada por:

b*((ab)*|a)

los senderos que conducen del estado inicial al final son:

- Estado final A: b*, b*(ab)* y al estado final B: b*a

La generación de la gramática a partir del autómata se la puede realizar mediante las siguientes conversiones:

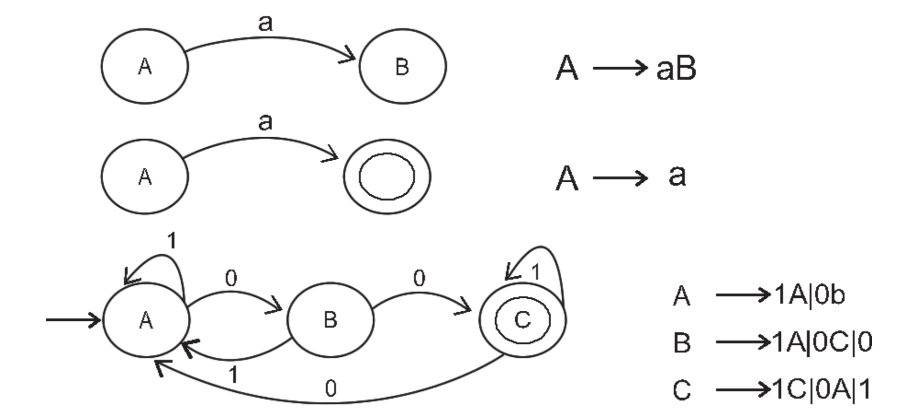


Figura 4.13: Conversiones de autómata a gramática regular

La gramática para el autómata de la Figura 4.11 que reconoce números múltiples de tres es:

A => 0A|1B|0

B => 1A|0C|1

C => 0B|1C

Es posible generar un autómata a partir de la gramática o expresión regular.

El autómata de estado finito es una herramienta para el análisis léxico; permite identificar los tokens de las sentencias; por ejemplo para identificadores:

<id> => <letra>(<letra>|<digito>)*

El símbolo "otro" representa un símbolo diferente a una letra o digito que actúa como separador de tokens; se utiliza una metodología similar para números, operadores, palabras reservadas, etc.

4.7. Analizador Descendiente Recursivo

El analizador descendente recursivo es una especie de analizador sintáctico de arriba hacia abajo construido en base a un conjunto de procedimientos mutuamente recursivos en el que cada procedimiento implementa una de las reglas de producción de la gramática.

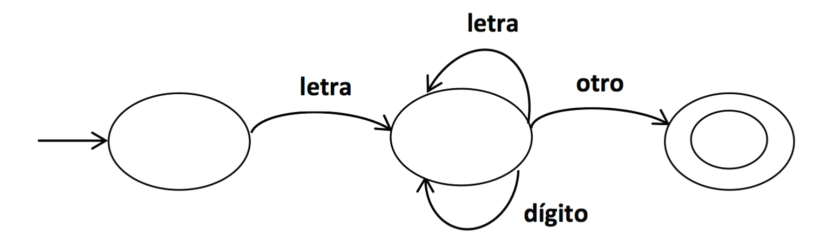


Figura 4.14: Autómata finito determinístico para la regla que define identificadores.

El analizador predictivo es el analizador descendente recursivo que no requiere retroceso. Este analizador es posible para las gramáticas LL(k) que son las de contexto libre excluyendo las ambiguas y las recursivas por la izquierda.

Para la siguiente gramática LL(1) :

```
program = block "." .

 block =
     ["const" ident "=" number {"," ident "=" number} ";"
        ]
     ["var" ident {"," ident} ";"]
     {"procedure" ident ";" block ";"} statement .

 statement =
     [ident ":=" expression
     | "call" ident
     | "begin" statement {";" statement} "end"
     | "if" condition "then" statement
     | "while" condition "do" statement
     ] .

 condition =
     "odd" expression
     | expression ("="|"#"|"<"|"<="|">"|">=") expression
        .
```

```
expression = ["+"|"-"] term {("+"|"-") term} .

term = factor {("*"|"/") factor} .

factor =
    ident
    | number
    | "(" expression ")" .
```

La implementación en C del analizador recursivo descendente es:

```
typedef enum {ident, number, lparen, rparen, times,
slash, plus, minus, eql, neq, lss, leq, gtr, geq,
callsym, beginsym, semicolon, endsym, ifsym, whilesym,
becomes, thensym, dosym, constsym, comma, varsym,
procsym, period, oddsym} Symbol;
Symbol sym;
void getsym(void);
void error(const char msg[]);
void expression(void);

int accept(Symbol s) {
    if (sym == s) {
        getsym();
        return 1;
    }
    return 0;
}

int expect(Symbol s) {
    if (accept(s))
        return 1;
    error("esperado: simbolo no esperado");
    return 0;
}

void factor(void) {
    if (accept(ident)) {
```

```
        ;
    } else if (accept(number)) {
        ;
    } else if (accept(lparen)) {
        expression();
        expect(rparen);
    } else {
        error("factor: error sintÃ¡ctico");
        getsym();
    }
}

void term(void) {
    factor();
    while (sym == times || sym == slash) {
        getsym();
        factor();
    }
}

void expression(void) {
  if (sym == plus || sym == minus)
        getsym();
    term();
    while (sym == plus || sym == minus) {
        getsym();
        term();
    }
}

void condition(void) {
    if (accept(oddsym)) {
        expression();
    } else {
        expression();
        if (sym == eql || sym == neq || sym == lss || sym
            == leq || sym == gtr || sym == geq) {
            getsym();
```

```
            expression();
        } else {
            error("condicion: operador invalido");
            getsym();
        }
    }
}

void statement(void) {
    if (accept(ident)) {
        expect(becomes);
        expression();
    } else if (accept(callsym)) {
        expect(ident);
    } else if (accept(beginsym)) {
        do {
            statement();
        } while (accept(semicolon));
        expect(endsym);
    } else if (accept(ifsym)) {
        condition();
        expect(thensym);
        statement();
    } else if (accept(whilesym)) {
        condition();
        expect(dosym);
        statement();
    }
}

void block(void) {
    if (accept(constsym)) {
        do {
            expect(ident);
            expect(eql);
            expect(number);
        } while (accept(comma));
        expect(semicolon);
    }
```

```
    if (accept(varsym)) {
        do {
            expect(ident);
        } while (accept(comma));
        expect(semicolon);
    }
    while (accept(procsym)) {
        expect(ident);
        expect(semicolon);
        block();
        expect(semicolon);
    }
    statement();
}

void program(void) {
    getsym();
    block();
    expect(period);
}
```

Nótese que el analizador predictivo es muy cercano a la gramática. Existe un procedimiento para cada no terminal de la gramática. El programa depende de la variable global sym que contiene el siguiente símbolo de la cadena de entrada, y la función getsym que actualiza sym cuando es llamada.

Capítulo 5

Semántica

La semántica se refiere a la interpretación o la comprensión de los programas y de como predecir el resultado de la ejecución del programa. La semántica de un lenguaje de programación describe la relación entre la sintaxis y el modelo de computación. La semántica se puede considerar como una función que mapea construcciones sintácticas al modelo computacional.

semántica: la sintaxis -> modelo computacional

Este enfoque se denomina semántica dirigida por la sintaxis.

Hay cuatro técnicas ampliamente utilizadas (algebraica, axiomática, denotacional, y operacional) para la descripción de la semántica de los lenguajes de programación.

Las semánticas algebraicas expresan el significado de un programa mediante la definición de un álgebra que describe relaciones algebraicas que existen entre los elementos sintácticos del lenguaje. Las relaciones algebraicas y las operaciones son descritas por los axiomas y las ecuaciones.

La semántica axiomática define el significado del programa de manera implícita. Se definen las propiedades de las construcciones del lenguaje. Estas propiedades se expresan con axiomas y reglas de inferencia. Las propiedades de un programa se deducen usando los axiomas y reglas de inferencia. Cada programa tiene unas pre-condiciones que describen las condiciones iniciales requeridas por el programa antes de su ejecución, y un conjunto de post-condiciones que describen, luego de su finalización, las propiedades deseadas del programa. Se hacen

afirmaciones sobre las relaciones que sostienen en cada punto en la ejecución del programa.

La semántica denotacional dice que es lo que se calcula describiendo un objeto matemático (por lo general una función) y que es el significado del programa. La semántica denotacional se utiliza en los estudios comparativos de lenguajes de programación.

La semántica operacional describe cómo un cálculo se realiza mediante la definición de como simular la ejecución del programa. La semántica operacional describe las transformaciones sintácticas que imitan la ejecución del programa en una máquina abstracta o define una traducción del programa en funciones recursivas. La semántica operacional se utiliza cuando se aprende un lenguaje de programación y es también utilizada por los diseñadores de compiladores.

Gran parte del trabajo en la semántica de los lenguajes de programación está motivado por los problemas que se plantean al tratar de construir y entender programas imperativos, programas con comandos de asignación. Dado que el comando de asignación reasigna valores a las variables, la asignación puede tener efectos inesperados en partes distantes del programa.

5.1. Semántica Algebraica

Una definición algebraica de un lenguaje es una definición de un álgebra. Un álgebra consiste de un dominio de valores y un conjunto de operaciones (funciones) definidas en el dominio.

Álgebra = <conjunto de valores, operaciones>

La siguiente definición formal (Cuadro 5.1) contiene un ejemplo de una definición algebraica. Es una definición algebraica de un fragmento de la aritmética de Peano. Las ecuaciones semánticas definen las equivalencias entre los elementos sintácticos. Las ecuaciones especifican las transformaciones que son usadas para traducir de una forma sintáctica a otra.

El dominio se llama a menudo "tipo" y el dominio y las secciones de funciones semánticas constituyen la "firma" del álgebra. Funciona con cero, uno y dos

```
Dominios:
Bool = {true, false} (Valores booleanos)
N en Nat (Los números naturales)
N ::= 0 | S(N)

Funciones:

= : (Nat, Nat) -> Bool
+ : (Nat, Nat) -> Nat
x : (Nat, Nat) -> Nat

Axiomas y Ecuaciones:

not S(N) = 0
if S(M) = S(N) then M = N
( n + 0 ) = n
( m + S(n) ) = S( m + n )
( n x 0 ) = 0
( m x S(n)) = (( m x n) + m)

donde m,n en Nat
```

Cuadro 5.1: Definición Algebraica de la Aritmética de Peano

operandos, las que se conocen como operaciones nulas, unarias y binarias, respectivamente. Frecuentemente, los tipos de datos abstractos requieren valores de diferentes dominios. Estos tipos son modelados usando álgebras multi-dominio. La firma del álgebra es un conjunto de dominios y un conjunto de funciones que toman argumentos y devuelven valores de diferentes dominios. Por ejemplo, una pila de números naturales puede ser modelada como un álgebra multi-dominio con tres dominios (los números naturales, pilas y booleanos) y cuatro operaciones (nuevaPila, push, pop, top, y vacia). La definición algebraica de una pila puede verse en la siguiente definición del Cuadro 5.2.

En el cuadro 5.1, se describe la estructura de los números. En la Figura 5.2 la estructura de una pila no se define. Esto significa que no podemos usar las ecuaciones para describir las transformaciones sintácticas. En cambio, usamos los axiomas que describen las relaciones entre las operaciones. Los axiomas son más abstractos que las ecuaciones, porque los resultados de las operaciones no

Dominios: Nat (los números naturales) Stack (de números naturales) Bool (valores booleanos) Funciones: newStack: () -> Stack push : (Nat, Stack) -> Stack pop: Stack -> Stack top: Stack -> Nat empty : Stack -> Bool Axiomas: o pop(push(N,S)) = S top(push(N,S)) = N empty(push(N,S)) = false empty(newStack()) = true Errores: pop(newStack()) top(newStack()) donde N en Nat y S en Stack.	Equaciones de Definición: newStack() = [] push(N,S) = [N|S] pop([N|S]) = S top([N|S]) = N

Cuadro 5.2: Definición Algebraica de una Pila de Enteros

se describen. Para ser más específicos, se requiere que se tomen decisiones sobre la aplicación de la estructura de datos pila. Estas decisiones tienden a oscurecer las propiedades algebraicas de las pilas. Los axiomas imponen limitaciones a las operaciones de la pila que son confiables, en el sentido de que son consistentes con el comportamiento real de las pilas independiente de la aplicación. Es más dificil encontrar axiomas que sea completos, en el sentido de que especifican completamente el comportamiento de las operaciones de un ADT.

El objetivo de la semántica algebraica es capturar la semántica de comportamiento de un conjunto de axiomas con propiedades puramente sintácticas. Definiciones algebraicas (álgebras semánticas) es el método preferido para la definición de las propiedades de los tipos de datos abstractos.

5.2. Semántica Axiomática

La semántica axiomática de un lenguaje de programación se refiere a las afirmaciones sobre las relaciones que se mantienen iguales, cada vez que se ejecuta el programa. Las semánticas axiomáticas son definidas para cada estructura de control y comando. La semántica axiomática de un lenguaje de programación define la teoría matemática de los programas escritos en el lenguaje.

Una teoría matemática tiene tres componentes:

- Las reglas sintácticas: Determinan la estructura de las fórmulas que son las declaraciones de interés.
- Axiomas: Estos describen las propiedades básicas del sistema.
- Las reglas de inferencia: Estos son los mecanismos para deducir nuevos teoremas a partir de axiomas y otros teoremas.

Las fórmulas semánticas son triples de la siguiente forma:

{P} c {Q}

donde c es un comando o una estructura de control en el lenguaje de programación, P y Q son afirmaciones o declaraciones sobre las propiedades de los objetos de programa (a menudo las variables del programa) que pueden ser verdaderas o falsas. P se conoce como una pre-condición y Q se conoce como una

post-condición. Las pre- y post- condiciones son las fórmulas en alguna lógica arbitraria y resumen el progreso del programa.

El significado de

{P} c {Q}

es que si c se ejecuta en un estado en que se cumple la afirmación de P y, c termina, entonces c termina en un estado en el que la afirmación de Q se satisface. Se ilustra la semántica axiomática con un programa para calcular la suma de los elementos de una matriz:

```
S,I := 0,0
 while I < n do
S,I := S+A[I+1],I+1
end
```

Las instrucciones de asignación son las declaraciones simultáneas de asignación. Las expresiones en el lado derecho se evalúan de forma simultánea y se asignan a las variables en el lado izquierdo en el orden en que aparecen.

La figura 5.1 ilustra el uso de la semántica axiomática para verificar el programa anterior.

El programa suma los valores almacenados en un arreglo y el programa contiene las afirmaciones que ayudan a verificar la corrección del código. La pre-condición en la línea 1 y la post-condición en la línea 11 son las pre- y post-condiciones, respectivamente, para el programa. La pre-condición afirma que el arreglo contiene al menos un elemento cero y que la suma de los primeros cero elementos de un arreglo es cero. La post-condición afirma que s es la suma de los valores almacenados en el arreglo. Después de la primera afirmación sabemos que la suma parcial es la suma de los primeros I elementos del arreglo y que es menor o igual al número de elementos en el arreglo.

La única manera de entrar a la malla del comando while es, si es el número de elementos sumados es menor que el número de elementos en el arreglo. Cuando éste es el caso, la suma de los primeros I +1 elementos del arreglo es igual a la suma de los primeros elementos de I más el elemento I+1 e I+1 es menor o igual a n. Después de la asignación en la malla del lazo, la afirmación de entrada del lazo se mantiene una vez más. Luego de la finalización del lazo, el índice del lazo es igual a n. Para demostrar que el programa es correcto, tenemos que demostrar

	Pre/Post-conditions	**Code**
1.	$\{ 0 = Sum_{i=1}{}^{0} A[i], 0 < \|A\| = n \}$	
2.		`S,I := 0,0`
3.	$\{s = Sum_{i=1}{}^{I} A[i], I <= n \}$	
4.		`while I < n do`
5.	$\{s = Sum_{i=1}{}^{I} A[i], I < n \}$	
6.	$\{s+A[I+1] = Sum_{i=1}{}^{I+1} A[i], I+1 <= n \}$	
7.		`S,I := S+A[I+1],I+1`
8.	$\{ S = Sum_{i=1}{}^{I} A[i], I <= n \}$	
9.		`end`
10.	$\{s = Sum_{i=1}{}^{I} A[i], I <= n, I >= n \}$	
11.	$\{s = Sum_{i=1}{}^{n} A[i] \}$	

Figura 5.1: Verificación de la suma S = i = 1 n A [i]

que las afirmaciones satisfacen algún sistema de verificación. Para verificar los comandos de asignación, se utiliza el Axioma de Asignación:

Axioma de Asignación:

$$\{P\ [x: E]\}\ x: = E\ \{P\}$$

Este axioma afirma que si después de la ejecución del comando de asignación, el entorno satisface la condición P, entonces el entorno antes de la ejecución del comando de asignación también satisface la condición P, pero con E sustituido por x (En este y los siguientes axiomas se supone que la evaluación de expresiones no produce efectos secundarios).

Un examen de las respectivas pre- y post-condiciones para las declaraciones de asignación demuestra que el axioma se cumple.

Para verificar el comando while de las líneas 4, 7 y 9, se utiliza el Axioma del Lazo:

Axioma del Lazo:

$$\{I \bigwedge B \bigwedge V > 0\}\ c\ \{I \bigwedge V > V\ '> = 0\}\ \{I\},$$

Siempre que $\{I\}$ while B do C end $\{I \bigwedge \neg B\}\}$

En esta regla, {I} es llamada la invariante del lazo. Este axioma afirma que para verificar un lazo, debe haber una invariante del lazo que es parte tanto de la pre- y post-condiciones del cuerpo del lazo y la expresión condicional del lazo debe ser cierta para ejecutar el cuerpo del lazo y falsa para salir del lazo.

La invariante del lazo es:

$$S = \sum_{i=1}^{I} A[i],\ I<=n$$

Las líneas 6, 7 y 8 satisfacen la condición para la aplicación del Axioma del Lazo. La demostración de la terminación requiere la existencia de una variante del lazo. La variante del lazo es una expresión cuyo valor es un número natural y cuyo valor es disminuido en cada iteración del lazo. La variante del lazo proporciona un límite superior al número de iteraciones del lazo.

Una variante de lazo es una expresión V con valor de número natural cuyo valor en tiempo de ejecución satisface las dos condiciones siguientes:

- Valor de V mayor que cero antes de cada ejecución del cuerpo del lazo.
- Ejecución del cuerpo del lazo disminuye el valor de V en por lo menos uno.

La variante de lazo para este ejemplo es la expresión n - I. El hecho de que no sea negativo está garantizado por la condición de la continuación del lazo y porque su valor se reduce en uno en el comando de asignación que se muestra en la línea 7. Variantes de lazo más generales se pueden utilizar; las variantes de lazo pueden ser expresiones en un conjunto bien definido (cada secuencia decreciente es finita). Sin embargo, no hay pérdida de generalidad al requerir que la expresión variante sea un entero. La recursividad se maneja de manera muy similar a los lazos, ya que debe haber una invariante y una variante.

5.2.1. Principio de Corrección de Lazos

Cada lazo debe tener una invariante y una variante.

Las líneas 5 y 6 y las líneas 10 y 11 se justifican por la Regla de las Consecuencias.

Regla de las Consecuencias:

P -> Q, C {Q} {R}, R -> S, S

Siempre que {P} c {S}

La justificación para la composición del comando de asignación en la línea 2 y el comando while, requieren el siguiente axioma de la composición secuencial.

Axioma de Composición Secuencial:

{P} {Q C0}, {Q} C1 {R},

Siempre que {P} C0; C1 {R}

Este axioma se lee de la siguiente manera: La composición secuencial de dos comandos se permite cuando la post-condición del primer comando es la pre-condición del segundo comando.

Las siguientes reglas son necesarias para completar el sistema deductivo:

Axioma de Selección:

{P $\bigwedge$ B} C0 {Q}, {P $\bigwedge\neg$ B} C1 {Q},

Siempre que {P} if B then C0, else C1 fi {Q}

Axioma de Conjunción:

{P} C {Q}, {P} C {Q '},

Siempre que {P $\bigwedge$ P} C {Q $\bigwedge$ Q '}

Axioma de Disyunción:

{P} C {Q}, {P} C {Q '},

Siempre {P $\bigvee$ P} C {Q $\bigvee$ Q '}

El método axiomático es el más abstracto de los métodos semánticos y, sin embargo, desde el punto de vista del programador, el método más práctico. Es más abstracto porque no trata de determinar el significado de un programa, sino sólo lo que puede ser probado sobre el programa. Esto hace que sea el más práctico ya que el programador tiene que preocuparse por cosas como si el programa terminara y qué tipo de valores se calcularán.

La semántica axiomática es apropiada para la verificación y derivación de programas.

5.2.2. Afirmaciones para la Construcción del Programa

Las técnicas axiomáticas pueden aplicarse a la construcción de software. En lugar de probar la exactitud de un programa existente, la prueba está integrada con el proceso de construcción de programas para asegurar la corrección desde el principio. Como el programa y la prueba se desarrollan conjuntamente, las afirmaciones pueden aportar sugerencias que faciliten la construcción de programas.

Los lazos y la recursividad son dos construcciones que requieren inventiva de parte del programador. El principio de corrección del lazo requiere que el programador defina tanto una variante y una invariante. La recursividad es una generalización de los lazos, así que las pruebas de corrección para los programas recursivos requieren también una variante y una invariante. En el ejemplo de la suma, una variante de lazo es fácilmente apreciable al examinar la postcondición. Basta con sustituir el límite superior de la suma, que es una constante, con una variable. Al inicializar la suma y el índice en cero se establece la invariante. Una vez que la invariante se ha establecido, ya sea que el índice sea igual al límite superior, en cuyo caso la suma se ha calculado o el siguiente valor debe

ser añadido a la suma, incrementando el índice restableciendo la invariante del lazo. La posición de los invariantes de lazo define el cuerpo del lazo y su segunda aparición sugiere una llamada recursiva. Una versión recursiva del programa de suma se da en el siguiente código:

```
S,I := 0,0
loop: if I < n then S,I := S+A[I+1],I+1; loop
          else skip fi
```

La ventaja de utilizar recursión es que la variante e invariante del lazo pueden ser desarrolladas por separado. En primer lugar se desarrolla la invariante, luego la variante.

El programa de suma se desarrolló a partir de la post-condición mediante la sustitución de una constante por una variable. La inicialización asigna un valor trivial a la variable para establecer la invariante y cada iteración del lazo mueve el valor de la variable hacia la constante.

Un programa para llevar a cabo la división entera por la resta repetida puede desarrollarse a partir de la post-condición $\{0 <= r < d, (a = q \times d + r)\}$ mediante la supresión de la conjunción. En este caso la invariante es $\{0 <= r, (a = q \times d + r)\}$ y se establece mediante la asignación de cero al cociente y el resto a a.

Otra técnica se utiliza para la construcción de programas con múltiples lazos. Por ejemplo, la post-condición de un programa de clasificación puede ser especificado como:

$$\{\forall\ i\ (0 < i < n \text{ -> } A\ [i] <= A\ [i+1]).,\ S = \text{permanente}\ (A)\}$$

o la post-condición de una rutina de búsqueda en un arreglo puede ser especificada como:

$$\{\text{if } i \text{ existe: } (0 < i <= N \text{ y } T = A\ [i]), \text{ then posición} = i \text{ else posición} = 0\}$$

Para desarrollar una invariante en estos casos se requiere que la afirmación sea reforzada mediante restricciones adicionales. Las restricciones adicionales hacen afirmaciones acerca de las diferentes partes del arreglo.

5.3. Semántica Denotacional

Una definición denotacional de un lenguaje se compone de tres partes: la sintaxis abstracta del lenguaje, un álgebra semántica que defina un modelo computacional, y las funciones de valoración. Las funciones de valoración mapean las construcciones sintácticas del lenguaje al álgebra semántica. La recursión y la iteración se definen mediante la noción de un límite. Las construcciones del lenguaje de programación se encuentran en el dominio sintáctico, mientras que la entidad matemática está en el dominio semántico y el mapeo entre los distintos dominios es proporcionado por las funciones de valoración. La semántica denotacional se basa en la definición de un objeto en términos de sus partes constitutivas. El Cuadro 5.3 es un ejemplo de una definición denotacional.

```
N en Nat (números naturales)
N ::= 0 | S(N) | (N + N) | (N x N)

Algebra Semántica:

Nat (los números naturales (0, 1, ...)
+ : Nat -> Nat -> Nat

Función de Valoración:

D : Nat -> Nat

D[( n + 0 )] = D[n]
D[( m + S(n) )] = D[(m+n)] + 1
D[( n x 0 )] = 0
D[( m x S(n))] = D[ (( m x n) + m) ]

donde m,n en Nat
```

Cuadro 5.3: Definición Denotacional de la Aritmética de Peano

Se trata de una definición denotativa de un fragmento de la aritmética de Peano. Nótese la sutil distinción entre los dominios sintácticos y semánticos. Las expresiones sintácticas se asignan a un álgebra de los números naturales por parte de

la función de valoración. La definición denotacional casi parece ser innecesaria, dado que la sintaxis se asemeja mucho al álgebra semántica.

Las definiciones denotacionales son favorecidas por los estudios teóricos y comparativos de los lenguajes de programación. Las definiciones denotacionales se han utilizado para la construcción automática de compiladores para lenguajes de programación.

Denotaciones que no sean los objetos matemáticos son posibles. Por ejemplo, el autor de un compilador preferiría que el objeto denotado sea código objeto apropiado. Se han desarrollado sistemas para la construcción automática de compiladores a partir de la especificación denotacional de un lenguaje de programación.

5.4. Semántica Operacional

Una definición operacional de un lenguaje se compone de dos partes: una sintaxis abstracta y un intérprete. El intérprete define la forma de realizar un cálculo. Cuando el intérprete evalúa un programa, genera una secuencia de configuraciones de máquina que definen la semántica operacional del programa. El intérprete es una relación de evaluación que se define por reglas de re-escritura. El intérprete puede ser una máquina abstracta o funciones recursivas. El cuadro 5.4 es un ejemplo de una definición operativa. Se trata de una definición operativa de un fragmento de la aritmética de Peano.

El intérprete es usado para reescribir las expresiones de números naturales a una forma estándar (una forma que implica solamente S y 0) y las reglas de reescritura muestran cómo se mueven los operadores + y × hacia adentro, hacia los casos base. Las definiciones operacionales son favorecidas por los implementadores de lenguajes para la construcción de compiladores y por tutoriales de lenguajes, porque las definiciones operacionales describen como las acciones se llevan a cabo.

La semántica operacional se define mediante dos funciones semánticas, l que interpreta los comandos y nu, que evalúa las expresiones. El intérprete es más complejo, ya que hay un ambiente asociado con el programa que no aparece

```
Sintaxis Abstracta:

N en Nat (números naturales)

N ::= 0 | S(N) | (N + N) | (N x N)

Interpretador:

I: N -> N

I[ ( n + 0 ) ]        ==> n
I[ ( m + S(n) ) ]     ==> S( I[ (m+n ) ] )
I[ ( n x 0 ) ]        ==> 0
I[ ( m x S(n)) ]      ==> I[ (( m x n) + m) ]


donde m,n en Nat
```

Cuadro 5.4: Semántica Operacional para la Aritmética de Peano

como un elemento sintáctico y, entorno es el resultado del cómputo. El entorno (llamado entorno referente) es una asociación entre las variables y los valores a los que están asignados. Inicialmente, el entorno está vacío ya que ninguna variable ha sido asignada a un valor. Durante la ejecución del programa, cada asignación actualiza el entorno. El intérprete tiene una función auxiliar que se utiliza para evaluar expresiones.

Las semánticas operacionales son particularmente útiles para construir una implementación de un lenguaje de programación.

Capítulo 6

Valores y Tipos de Datos

¿Qué es un valor?

Un valor es cualquier cosa que pueda ser evaluada, guardada, incorporada en una estructura de datos, pasada como argumento o retornada como resultado.

¿Qué es un tipo?

- Realista: Un tipo es un conjunto de valores.
- Idealista: No. Un tipo es una entidad conceptual cuyos valores son accesibles sólo a través del filtro interpretativo de tipo.
- Programador principiante: ¿No es un tipo un nombre para un conjunto de valores?
- Programador intermedio: Un tipo es un conjunto de valores y operaciones.
- Programador avanzado: Un tipo es una manera de clasificar los valores por sus propiedades y comportamiento.
- Algebrista: ¡Ah! un tipo es un álgebra, un conjunto de valores y operaciones definidas en esos valores.
- Revisor de tipos: Los tipos son más prácticos que eso, son restricciones en las expresiones para asegurar la compatibilidad entre los operadores y su(s) operando(s).

- Sistema de inferencia de tipos: Sí y más, puesto que un sistema tipo es un conjunto de reglas para asociar, con cada expresión, un tipo único y más general que refleja el conjunto de todos los contextos significativos en los que la expresión puede ocurrir.
- Verificador de programa: Vamos a mantenerlo simple, los tipos son las invariantes del comportamiento que las instancias del tipo deben cumplir.
- Ingeniero de software: Lo que es importante para mí es que los tipos son una herramienta para la gestión del desarrollo y evolución del software.
- Compilador: Toda esta charla me confunde, los tipos especifican los requerimientos de almacenamiento para las variables del cada tipo.

Cómputo es una secuencia de operaciones aplicadas a un valor para obtener otro valor. Los valores y operaciones son fundamentales para el cómputo. Los valores son el tema de este capítulo y las operaciones son objeto de los siguientes capítulos.

En terminología matemática, los conjuntos de los cuales los argumentos y los resultados de una función se toman, son conocidos como "dominio" y "codominio" de la función, respectivamente. En consecuencia, el término dominio denotará cualquier conjunto de valores que pueden ser pasados como argumentos o devueltos como resultados. Asociado con cada dominio están ciertas "operaciones esenciales". Por ejemplo, el dominio de los números naturales está equipado con operación "constante" que produce el número cero y la operación que construye el sucesor de cualquier número. Operaciones adicionales (como la suma y multiplicación) de los números naturales se pueden definir con estas operaciones básicas.

Los lenguajes de programación utilizan un amplio conjunto de dominios. Valores de verdad, caracteres, enteros, reales, registros, arreglos, conjuntos, archivos, punteros, abstracciones de procedimientos y funciones, entornos, comandos y definiciones, no son sino algunos de los dominios que se encuentran en los lenguajes de programación. Hay dos acercamientos a los dominios. Un método consiste en asumir la existencia de un dominio universal. Este dominio universal contiene todos los objetos que son de interés computacional. El segundo método es comenzar con un pequeño conjunto de valores y algunas reglas para la combinación de estos valores para la construcción del universo de valores. Los lenguajes de programación siguen el segundo enfoque, proporcionando varios conjuntos

básicos de valores y un conjunto de constructores de dominio, para que cada uno de los dominios adicionales puedan ser construidos.

Los dominios se clasifican como primitivo o compuesto. Un dominio primitivo es un conjunto que es fundamental para la aplicación que se está estudiando. Sus elementos son atómicos. Un dominio compuesto es un conjunto cuyos valores se construyen a partir de los dominios existentes por uno o más constructores de dominio.

6.1. Teoría de Dominios

La teoría de dominios es el estudio de conjuntos estructurados y sus operaciones. Un dominio es un conjunto de elementos y un conjunto de operaciones definidas en el dominio.

Los términos dominio, tipo y tipo de datos se pueden usar indistintamente.

El término datos se refiere tanto a un elemento de un dominio o una colección de elementos de uno o más dominios.

Los términos compuesto y estructurado, cuando se aplica a los valores, datos, dominios y tipos, se utilizan indistintamente.

Hay muchos dominios compuestos que son útiles en ciencias de la computación: arreglos, tuplas, registros, variantes, uniones, conjunto, listas, árboles, archivos, relaciones, definiciones, mapeos, etc., son todos ejemplos de dominios compuestos. Cada uno de estos dominios puede ser construido a partir de dominios más simples por una o más llamadas a los constructores de dominio.

Los dominios compuestos son construidos por un constructor de dominio. Un constructor de dominio se compone de un conjunto de operaciones de montaje y desmontaje de elementos de un dominio compuesto. Los constructores de dominio son:

- Dominio Producto
- Dominio Suma
- Dominio Función
- Dominio Potencia
- Dominio Recursivo

6.1.1. Dominio Producto

Los dominios construidos por el constructor dominio producto son llamados tuplas en ML, registros en Cobol, Pascal y Ada, y estructuras en C y C++. Los dominios producto constituyen el fundamento de las bases de datos relacionales y de la programación lógica.

En el caso binario, el constructor de dominio producto, x, construye el dominio A x B, a partir de los dominios A y B. El constructor de dominio incluye la operación de ensamblaje (constructor de par ordenado) y un conjunto de operaciones de desmontaje denominadas funciones de proyección. La operación de montaje, constructor de par ordenado, se define como sigue:

Si a es un elemento de A y b es un elemento de B entonces (a,b) es un elemento de A x B. Es decir,

$$\mathrm{A\ x\ B} = \{(\mathrm{a,b}) \mid \mathrm{a\ en\ A,\ b\ en\ B}\}$$

Las operaciones de desensamblaje fst y snd son funciones de proyección que extraen elementos de las tuplas. Por ejemplo, fst extrae el primer componente y snd extrae el segundo elemento

$$\mathrm{snd}(\mathrm{a},\ \mathrm{b}) = \mathrm{b}$$

El dominio producto es fácilmente generalizado para construir el producto de un número arbitrario de dominios.

Ensamblaje:

$(a_0,...,a_n)$ en D_0 x ... x D_n donde a_i en D_i
y D_0 x ... x D_n = { $(a_0,...,a_n)$ | a_i en D_i }

Desensamblaje:

$(a_0,...,a_n)|_i = a_i$ para $0 <= i <= n$

Tanto las bases de datos relacionales, como el paradigma de programación lógica (Prolog) se basan en la programación con tuplas.

Los elementos de dominios producto normalmente son implementados como un bloque contiguo de almacenamiento en el que los componentes se almacenan en secuencia. La selección de componentes está determinada por un desplazamiento desde la dirección de la primera unidad de almacenamiento del bloque de almacenamiento. Una implementación alternativa (que puede requerirse en los lenguajes de programación funcional o lógica) es implementar el valor como una lista de valores. La selección de componentes utiliza las operaciones de lista disponibles.

El dominio producto también se conoce como el producto cartesiano o producto cruz.

6.1.2. Dominio Suma

Los dominios construidos por el constructor de dominio suma son llamados registros variantes en Pascal y Ada, uniones en Algol-68, construcciones en ML y tipos algebraicos en Miranda.

En el caso binario, el constructor de dominio suma, +, construye el dominio A + B de los dominios A y B. El constructor de dominio incluye un par de operaciones de ensamblaje y una operación de desmontaje. Las dos operaciones de ensamblaje del constructor suma se definen como sigue:

Si a es un elemento de A y b es un elemento de B entonces (A,a) y (B,b) son elementos de A +B. Esto es,

$$\mathrm{A} + \mathrm{B} = \{(\mathrm{A}, \mathrm{a}) \mid \mathrm{a\ en\ A}\}\ \mathrm{unión}\ \{(\mathrm{B}, \mathrm{b}) \mid \mathrm{b\ en\ B}\}$$

donde A y B son llamados etiquetas y se utilizan para distinguir entre los elementos aportados por A y los elementos aportados por B.

La operación de desmontaje devuelve el elemento si y sólo si la etiqueta coincide con la solicitud.

$$\mathrm{A}\ (\mathrm{A}, \mathrm{a}) = \mathrm{a}$$

El dominio suma difiere de la unión de conjuntos ordinaria en que los elementos de la unión están etiquetados con el conjunto padre. Así, aun cuando dos conjuntos contengan el mismo elemento, las etiquetas del constructor de dominio suma los etiqueta diferente.

El dominio suma se generaliza a sumas de un número de dominios arbitrarios.

Ensamblaje:

$$(D_i,\ d_i) \text{ en } D_0 + ... + D_n \text{ y } D_0 + ... + D_n = \text{Union}_{i=0}{}^{n}\ \{\ (D_i,d)\ |\ d \text{ en } D_i\ \}$$

Desensamblaje:

$$D_i(D_i,\ d_i) = d_i$$

El dominio suma también es llamado dominio de unión disjunta o dominio coproducto.

Los elementos del dominio suma suelen ser implementados en una parte contigua de memoria, lo suficientemente grande como para contener un valor de cualquiera de los dominios y, una etiqueta que se utiliza para determinar el dominio al que pertenece el valor.

6.1.3. Dominio Función

Los dominios construidos por el constructor de dominio función se llaman funciones en Haskell, procedimientos en Modula-3, y procs en SR. A pesar de su sintaxis a menudo difiere de la de las funciones, los arreglos también son ejemplos de dominios construidos por el constructor de dominio función.

El constructor de dominio función crea el dominio A -> B a partir de los dominios A y B. El dominio A -> B consta de todas las funciones de A a B. A se conoce como el dominio y B se llama el co-dominio.

La operación de ensamblaje es:

(lambda x, e) es un elemento de A -> B, siempre que e es una expresión que contiene las ocurrencias de un identificador x, tal que, cuando un valor a en A remplaza las ocurrencias de x en e, resulta el valor de e[a: x] en B.

La operación de desmontaje es la aplicación de función. Tiene dos argumentos, un elemento f de A -> B y un elemento a de A y produce f (a), un elemento de B. En el caso de los arreglos, la operación de desmontaje se llama subíndice.

Ensamblaje:

(lambda x.E) en A –> B donde para todo a en A, E[x:a] es un valor único en B.

Desensamblaje:

(g a) en B, para g en A –> B y a en A.

Las asignaciones (o funciones) de un conjunto a otro es un método de composición muy importante. El mapeo m de un elemento x de S (llamado el dominio) al elemento correspondiente m (x) de T (llamado el rango) se escribe como:

m: S -> T

donde si m(x) = a y m(y) = a, entonces x = y.

Los mapeos son más restringidos que el producto cartesiano, ya que, para cada elemento del dominio hay un único elemento en el rango. Muchas veces es, ya sea, difícil de especificar el dominio de una función o sino, la implementación no soporta el dominio o rango completos de una función. En tales casos, la función se dice que es una función parcial. Es por razones de eficiencia que las funciones parciales son permitidas y se convierte, en responsabilidad del programador informar a los usuarios del programa de la naturaleza, de la falta de fiabilidad.

Los arreglos son mapeos de un conjunto de índices a un tipo de elemento del arreglo. Un arreglo es un mapeo finito. Aparte de los arreglos, los mapeos se presentan como operaciones y abstracciones de funciones. Los valores del arreglo

se implementan mediante la asignación de un bloque contiguo de almacenamiento, donde el tamaño de un bloque está basado en el producto del tamaño del elemento del arreglo y el número de elementos del arreglo.

Las operaciones provistas para los tipos primitivos son mapas. Por ejemplo, la operación de adición es una aplicación del producto cartesiano de los números a los números:

+: número x número -> número

El paradigma de programación funcional se basa en la programación con mapas.

El dominio función también se conoce como el espacio de funciones.

Los elementos del dominio función son, usualmente, implementados en código. Sin embargo, los arreglos son un caso especial de dominio función y se implementan normalmente en elementos de memoria contiguos.

6.1.4. Dominio Potencia

La teoría de conjuntos proporciona una notación elegante para la descripción de la computación; sin embargo, es difícil proporcionar una aplicación eficaz de las operaciones de conjuntos. SETL es un lenguaje de programación basado en series y fue utilizado para proporcionar un compilador temprano para Ada. La familia de los lenguajes Pascal proveen unión de conjuntos y la intersección y la pertenencia a grupos. Las variables establecidas representan subconjuntos de los conjuntos definidos por el usuario.

El conjunto de todos los subconjuntos de un conjunto es el conjunto potencia y se define de la siguiente manera:

$$\mathrm{P}^{\mathrm{S}} = \{\mathrm{S} \mid \text{s es un subconjunto de S}\}$$

Subtipos y subrangos son ejemplos del constructor de conjuntos potencia.

Las funciones son subconjuntos de dominios potencia. Por ejemplo, la función cuadrado se puede representar como un subconjunto del dominio producto Nat x Nat.

$$\text{sqr} = \{(0,0),\ (1,1),\ (2,4),\ (3,9),\ ...\}$$

La generalización ayuda a simplificar esta lista infinita:

sqr = {(x, x * x) | x en Nat}

El lenguaje de programación SETL se basa en la computación a través de conjuntos.

Los valores del conjunto se pueden implementar utilizando el hardware subyacente para cadenas de bits. Esto hace que las operaciones de conjuntos sean eficientes, pero restringe el tamaño de los conjuntos para el número de bits (normalmente) en una palabra de almacenamiento. Alternativamente, los valores del conjunto pueden ser implementados mediante software, en cuyo caso, se puede utilizar códigos de hash o listas.

Algunos lenguajes proveen mecanismos para descomponer un tipo en subtipos:

- Enumeración de los elementos del subtipo.
- Subintervalos, ya que la enumeración es tediosa para sub-dominios grandes y muchos de ellos tienen un ordenamiento natural.

El constructor de dominio potencia construye un conjunto de elementos. Para un dominio A, el constructor de dominio potencia P() crea el dominio P(A), una colección cuyos miembros son subconjuntos de A:

Ensamblaje:

Ø en P^D, { a } en P^D para a en D, y S_i unión S_j en P^D para S_i, S_j en P^D

6.1.5. Dominio Recursivo

Los dominios recursivos son aquellos dominios de la forma:

D : ... D ...

La definición se llama recursiva porque el nombre del dominio "se repite" en el lado derecho de la definición. Los dominios recursivos dependen de la abstracción, ya que el nombre del dominio es una parte esencial de la definición del

dominio. Las gramáticas libres de contexto utilizadas en la definición de lenguajes de programación contienen definiciones recursivas, así que los lenguajes de programación son ejemplos de tipos recursivos.

Más de un conjunto puede satisfacer una definición recursiva. Sin embargo, puede demostrarse que una definición recursiva tiene al menos una solución. Esta solución es un subconjunto de todas las otras soluciones.

La solución menor de un dominio definido de forma recursiva se obtiene a través de una secuencia de aproximaciones (D_0, D_1, ...) al dominio, siendo el dominio el límite de la secuencia de aproximaciones (D = $\lim_{i->infinito}$ Di). El límite es la solución más pequeña a la definición de dominio recursivo.

$$\mathrm{D_0} = \mathrm{null}$$

$$\mathrm{D_{i+1}} = \mathrm{e[D{:}D_i]\ para\ i{=}0,...}$$

$$\mathrm{D} = \mathrm{lim_{i\ ->\ infinito}\ D_i}$$

Dado que los dominios recursivos como listas, pilas y los árboles no están acotados (en general, pueden ser objetos infinitos), se implementan usando dominio producto donde un dominio es un nodo y uno o más son dominios de direcciones. En Pascal, Ada y C, tales dominios se definen en términos de los punteros mientras que lenguajes como Prolog y lenguajes funcionales como ML y Miranda permiten definir tipos recursivos directamente.

6.2. Tipos Abstractos

Un tipo abstracto es un tipo que se define por sus operaciones en lugar de sus valores.

Los tipos de datos proporcionados en los lenguajes de programación son tipos abstractos. Por ejemplo, la representación del tipo entero está oculta para el programador.

El programador dispone de un conjunto de operaciones y una representación de alto nivel de los números enteros. El programador sólo se da cuenta del nivel más bajo cuando ocurre un desborde aritmético.

Un tipo abstracto de datos consiste en un nombre del tipo y las operaciones para crear y manipular objetos del tipo. Una idea clave es la separación de la implementación de la definición de tipo. El formato real de los datos está oculto (ocultamiento de la información) para el usuario y el usuario obtiene acceso a los datos sólo a través de las operaciones de tipo.

Hay dos ventajas a la definición de un tipo abstracto como un conjunto de operaciones. En primer lugar, la separación de las operaciones de la representación da como resultado la independencia de datos. En segundo lugar, las operaciones se pueden definir en una forma matemáticamente rigurosa. Como se indica en el capítulo sobre Semántica, las definiciones algebraicas proporcionan un método apropiado para definir un tipo abstracto. La especificación formal de un tipo abstracto se puede separar en dos partes. Una especificación sintáctica que ofrece la firma de las operaciones y una parte semántica en el que axiomas describen las propiedades de las operaciones.

Para ser totalmente abstracto, el usuario del tipo abstracto no debe tener acceso a la representación de los valores del tipo. Este es el caso con los tipos primitivos. Por ejemplo, los números enteros podrían estar representados en números binarios en complemento de dos, pero no hay manera de averiguar la representación sin salirse del lenguaje. Por lo tanto, un concepto clave de tipos abstractos es la ocultación de la representación de los valores del tipo. Esto significa que la información de la representación debe ser local a la definición del tipo.

El enfoque de Modula-3 es típico. Un tipo abstracto se define a partir de módulos: la definición de un módulo (llamada una interfaz) y la implementación del módulo (llamada módulo).

Dado que la representación de los valores del tipo está escondida, los tipos abstractos deben contar con las operaciones de constructor y destructor. Una operación de constructor crea un valor del tipo a partir de valores de algún otro tipo o tipos, mientras una operación destructor extrae un valor constituyente de un tipo abstracto. Por ejemplo, un tipo abstracto de los números racionales podría representar números racionales como pares de números enteros. Esto significa que la definición del tipo abstracto incluiría una operación a la que se da un par de enteros y devuelve un número racional (cuya representación como un par ordenado está oculta), que corresponde al cociente de los dos números. El aditivo racional y las identidades multiplicativas correspondientes a cero y uno también se proveen.

A los tipos abstractos también se los conoce como ADT o tipos de datos abstractos.

6.3. Sistemas de Tipos

Un gran porcentaje de errores en los programas se debe a que la aplicación de las operaciones a los objetos de tipos son incompatibles. Los sistemas de tipos han sido desarrollados para ayudar al programador en la detección de estos errores. Un sistema de tipos es un conjunto de reglas para definir los tipos y la asociación de un tipo con una expresión en el lenguaje. Un sistema de tipos rechaza una expresión si no asocia un tipo con la expresión. La verificación de tipos puede realizarse en tiempo de compilación o en tiempo de ejecución o en ambas.

Si se desea detectar los errores en tiempo de compilación entonces se requiere un sistema estático de verificación de tipo. Un enfoque para la verificación estática de tipos es exigir que el programador especifique el tipo de cada objeto en el programa. Esto permite que el compilador realice la comprobación de tipos antes de la ejecución del programa y este es el enfoque adoptado por lenguajes como Pascal, Ada, C++ y Java. Otro enfoque para la verificación estática de tipos es agregar capacidades de inferencia de tipos en el compilador. En tal sistema, el compilador realiza la verificación de tipos por medio de un conjunto de reglas de inferencia de tipo y es capaz de marcar errores de tipo antes del tiempo de ejecución. Este es el enfoque adoptado por Miranda y Haskell.

Si la detección de errores se va a llevar a cabo en el tiempo de ejecución, se necesita una verificación dinámica de tipos. En la verificación dinámica de tipos, cada valor de datos se etiqueta con la información de tipo para que, el entorno de tiempo de ejecución pueda comprobar la compatibilidad del tipo y posiblemente, realizar conversiones de tipos si es necesario. Los lenguajes de programación Lisp, Scheme y Smalltalk son ejemplos de lenguajes con tipos dinámicos.

6.3.1. Verificación de Tipos

Las operaciones de máquina manipulan patrones de bits. Ya sea que el patrón de bits represente un caracter, un número entero, un real, una dirección, o una instrucción, cualquier operación de la máquina se puede aplicar a cualquier elemento de datos. No hay ninguna verificación de tipos al nivel del lenguaje ensamblador. Los lenguajes que permiten que las operaciones se apliquen a cualquier tipo de dato se denominan sin-tipos. Prolog es uno de los pocos lenguajes de alto nivel que es un lenguaje sin-tipos. En Prolog, las listas puede consistir de elementos de cualquier tipo y diferentes tipos de valores puede ser comparados con la relación de igualdad "=", pero dicha comparación produce siempre falso.

En C, la condición de cualquier estructura de control puede ser cualquier expresión que produzca un valor. Si el valor es 0, se trata como falso y los valores distintos de cero se tratan como verdadero. Puesto que el valor de un comando de asignación es el valor de su lado derecho, en el comando if x = 4 ... cualquier cláusula else será ignorada. Los caracteres de C se tratan como enteros y por lo tanto pueden estar presentes en las expresiones aritméticas. El sistema de tipos de C no es lo suficientemente robusto como para proteger a los programadores principiantes de éstos y otros errores.

La ventaja de los lenguajes sin-tipo es su flexibilidad. El programador tiene control completo sobre cómo un valor de datos se utiliza, pero debe asumir la responsabilidad total de la detección de la aplicación de las operaciones a los objetos de tipo incompatible.

Un lenguaje se dice que es:

- sin-tipo o tipo-débil si no se hacen cumplir las abstracciones de tipo
- tipo-fuerte si se hace cumplir las abstracciones de tipo (las operaciones solo pueden aplicarse a los objetos del tipo apropiado)
- tipo-estático si el tipo de cada expresión puede ser determinado por el texto del programa al momento de compilar
- tipo-dinámico si la determinación del tipo de alguna expresión depende del comportamiento del programa en tiempo de ejecución.

Un lenguaje de tipo-fuerte impone abstracciones tipo. La mayoría de los lenguajes son de tipo fuerte con respecto a los tipos primitivos soportados por el lenguaje. Así, por ejemplo, la mezcla de tipos numéricos y caracteres que es permisible en C, no está permitido en Pascal o Ada.

La tipificación fuerte ayuda a asegurar la seguridad y la portabilidad del código y requiere, a menudo, que el programador defina explícitamente los tipos de cada uno de los objeto en un programa. También es importante en la compilación para escoger las operaciones apropiadas y para la optimización.

Si los tipos de todas las variables pueden ser conocidos a partir de un examen del texto (es decir, en tiempo de compilación), entonces el lenguaje se dice que es de tipo-estático. Pascal, Ada y Haskell son ejemplos de lenguajes de tipificación estática.

La tipificación estática es ampliamente reconocida como un requisito para la producción de software seguro y fiable. La verificación estática de tipos implica que los tipos se comprueban en tiempo de compilación. Los lenguajes de tipo-estático se eligen cuando la eficiencia en tiempo de ejecución es importante y se utiliza el apoyo del compilador para contribuir a las buenas prácticas de ingeniería de software.

Si el tipo de una variable sólo puede ser conocido en tiempo de ejecución, entonces el lenguaje se dice que es de tipo-dinámico. Lisp y Smalltalk son ejemplos de lenguajes tipificados dinámicamente.

La verificación dinámica de tipos implica que los tipos se comprueban en tiempo de ejecución y que cada valor está etiquetado para identificar su tipo con el fin de hacer posible la verificación de tipos. El costo de la verificación dinámica de tipos es la sobrecarga de tiempo y espacio adicionales.

El tipificado dinámico a menudo se justifica en el supuesto de que su flexibilidad permite la creación rápida de prototipos de software.

Prolog se basa en la comparación de patrones para proporcionar una apariencia de comprobación de tipos. Hay investigación activa en la adaptación de los sistemas de comprobación de tipos para Prolog.

Los lenguajes funcionales modernos como Haskell y Miranda y los lenguajes orientados a objetos, combinan la seguridad de verificación estática de tipos con la flexibilidad de la verificación dinámica, a través de los tipos polimórficos.

6.3.2. Equivalencia de Tipos

Dos tipos sin nombre (conjuntos de objetos) son los mismos si contienen los mismos elementos. Lo mismo no puede decirse de tipos con nombre porque si así fuera, entonces no habría necesidad del tipo unión disjunta. Cuando los tipos son nombrados, existen dos enfoques principales para determinar si dos tipos son iguales.

6.3.2.1. Equivalencia de Nombre

En la equivalencia de nombre, dos tipos son los mismos si tienen el mismo nombre. Los tipos que tienen diferentes nombres se tratan como si fueran distintos

y no pueden ser mezclados accidentalmente aunque su estructura sea la misma. La equivalencia de nombre requiere que las definiciones de tipo sean globales.

La equivalencia de nombre fue elegida para Modula-2, Ada, C (para los registros) y Miranda. El predecesor de Modula-2, Pascal viola la equivalencia de nombres ya que los nombres de tipos de archivos no están obligados a ser compartidos por diferentes programas de acceso al mismo archivo.

6.3.2.2. Equivalencia Estructural

En la equivalencia estructural, los nombres de los tipos se pasan por alto y los elementos de los tipos se comparan para determinar la igualdad. Es posible que dos tipos, lógicamente diferentes, puedan llegar a ser el mismo por casualidad y puedan ser mezclados. Las definiciones de tipos no están obligadas a ser globales. La equivalencia estructural es importante en la programación de sistemas distribuidos, en los cuales los distintos programas deben comunicarse datos tipificados.

- Dos tipos T, T' son nombre equivalente si y sólo si T y T' son el mismo nombre.
- Dos tipos T, T' son estructuralmente equivalentes si y sólo si T y T' tienen el mismo conjunto de valores.

Las siguientes tres reglas pueden ser utilizadas para determinar si dos tipos son estructuralmente equivalentes.

- Un tipo con nombre es estructuralmente equivalente a si mismo.
- Dos tipos son estructuralmente equivalentes si se crean por la aplicación del mismo constructor de tipo (recursivamente) a tipos estructuralmente equivalentes.
- Después de una declaración de tipo, tipo n =T, el nombre del tipo n es estructuralmente equivalente a T.

La equivalencia estructural fue elegida por Algol-68 y C (a excepción de los registros), ya que es fácil de implementar.

6.3.3. Inferencia de Tipos

La inferencia de tipos es el problema general de la transformación de la sintaxis sin-tipo o tipificada parcialmente en términos bien tipificados. Las declaraciones de constantes en Pascal son un ejemplo de inferencia de tipos, el tipo del nombre se deduce del tipo de la constante. En el lazo for de Pascal, el tipo del índice del lazo se puede deducir de los tipos de los límites del lazo y por lo tanto el índice del lazo debe ser una variable local al lazo. Los lenguajes de programación ML, Miranda y Haskell son tipo-estático y proporcionan potentes sistemas de inferencia de tipos, para que el programador no tenga que declarar los tipos. Estos lenguajes también permiten a los programadores proporcionar especificaciones explícitas de especificaciones de tipo.

Un verificador de tipo debe ser capaz de:

- determinar si un programa está bien tipificado y
- si el programa está bien tipificado, determinar el tipo de cualquier expresión en el programa

6.3.4. Declaraciones de Tipos

Aún los lenguajes que proporcionan sistemas de inferencia de tipos permiten a los programadores hacer declaraciones explícitas de tipo. Incluso si el compilador puede correctamente inferir los tipos, los lectores humanos pueden tener que revisar varias páginas de código para determinar el tipo de una función. Pequeños errores del programador pueden hacer que el compilador emita mensajes de error oscuros o que infiera un tipo diferente al previsto. Por estas razones, es buena práctica de programación declarar de forma explícita los tipos en todos los casos menos en los más obvios.

En Miranda (un lenguaje funcional), los tipos de la operación aritmética + se declaran como sigue:

```
+ :: num -> num -> num
```

En Pascal, el tipo de una función para calcular la circunferencia de un círculo se declara como sigue:

```
function circunferencia (radio: real): real;
```

6.3.5. Polimorfismo

El sistema de verificación de tipos es monomórfico si cada constante, parámetro, variable y resultado de la función tiene un tipo único. La comprobación de tipos en un sistema monomórfico es sencilla. Pero los sistemas de tipo puramente monomórficos no son satisfactorios para la escritura de software reutilizable. Muchos algoritmos, como la ordenación y la manipulación de listas y árboles son genéricos en el sentido de que dependen muy poco del tipo de los valores que están siendo manipulados. Por ejemplo, la rutina de ordenamiento de un arreglo de propósito general no se puede escribir en Pascal. Pascal requiere que el tipo de elemento del arreglo forme parte de la declaración de la rutina. Esto significa que diferentes rutinas de ordenamiento deban ser escritas para cada tipo de elemento y tamaño del arreglo.

Los sistemas completamente monomórficos son raros. La mayoría de los lenguajes de programación contienen algunos operadores o procedimientos que permiten argumentos de más de un tipo. Por ejemplo, los procedimientos de entrada y salida en Pascal permiten la variación, tanto en tipo como en número, de los argumentos. Este es un ejemplo de sobrecarga.

- Monomórfico: cada constante, variable, parámetro, el operador y la función tiene un tipo único.
- Sobrecarga: se refiere al uso de un identificador sintáctico único para referirse a varias operaciones diferentes, discriminadas por el tipo y número de los argumentos de la operación.
- Polimorfismo: un operador, función o procedimiento que tiene una familia de tipos relacionados y opera de manera uniforme en sus argumentos independientemente del tipo.
- Operación polimórfica: es una operación que se puede aplicar a argumentos de tipos diferentes pero relacionados.

El tipo de la operación suma definida para la adición entera es:

+: int x int -> int

Cuando el mismo símbolo de esta operación se utiliza para la operación suma de números racionales y para la unión de conjuntos, como en Pascal, el símbolo está sobrecargado. La mayoría de los lenguajes de programación prevé la sobrecarga

de los operadores aritméticos. Pocos lenguajes de programación (Ada, entre otros) permiten que el programador defina la sobrecarga tanto de los operadores integrados, así como aquellos definidos por el programador.

Cuando los operadores sobrecargados se aplican a expresiones mixtas como la suma a un número entero y a un número racional, hay dos opciones posibles; o bien la evaluación de la expresión falla o una o más de las subexpresiones son coercionadas en el objeto correspondiente de otro tipo. A los números enteros se los coerciona en números racionales correspondientes. Este tipo de coerción se denomina ampliación (widening). Cuando un lenguaje permite la coerción de un número real en uno entero (por truncamiento por ejemplo) la coerción se denomina estrechamiento (narrowing). El estrechamiento no se permite generalmente en los lenguajes de programación ya que, usualmente, la información se pierde. La coerción es un problema en lenguajes de programación porque los números no tienen una representación uniforme. Este tipo de sobrecarga se denomina sobrecarga dependiente del contexto.

Muchos lenguajes proporcionan funciones de tipo de transferencia para que el programador pueda explicitamente controlar dónde y cuándo se realiza la coerción de tipos. Truncar y redondear, son ejemplos de funciones de transferencia tipo.

La sobrecarga a veces se denomina polimorfismo ad-hoc.

La mayoría de los algoritmos de ordenamiento pueden ser explicados sin hacer referencia a la clase (tipo) de los datos que son ordenados. Por lo general, los datos son un arreglo de punteros a los registros, cada uno con una clave asociada. El tipo de la clave no importa, siempre y cuando exista un procedimiento de "comparación" que encuentre el mínimo entre un par de claves. Los procedimientos de ordenamiento utilizan la comparación de dos claves usando el procedimiento de comparación y luego intercambian los registros al restablecer los punteros. Sin embargo, en un lenguaje fuertemente tipificado esto no es posible, ya que el tipo de puntero depende del tipo de registro. Esto obliga a escribir un procedimiento separado para cada tipo de datos.

Pilas, colas, listas y árboles también son, en gran parte, independientes de tipo y, sin embargo, en un lenguaje fuertemente tipificado, el código debe ser escrito por separado para cada tipo de elemento. Algunos lenguajes permiten variables de tipo y estas estructuras de datos se pueden definir con una variable de tipo que luego especifique el usuario.

Un sistema de tipos es polimórfico si las abstracciones operan de manera uniforme en los argumentos de una familia de tipos relacionados.

Este tipo de polimorfismo se conoce como polimorfismo paramétrico.

La generalización se puede aplicar a muchos aspectos de los lenguajes de programación. A veces hay varios dominios que comparten una operación común. Por ejemplo, los números naturales, los enteros, racionales y los reales comparten la operación de adición. Por lo tanto, la mayoría de los lenguajes de programación utilizan el mismo operador de suma para indicar la suma en todos estos dominios. Pascal extiende el uso del operador de suma para representar la unión de conjuntos. El uso múltiple de un nombre en diferentes dominios se denomina sobrecarga.

Mientras que la parametrización de un objeto da la capacidad para hacer frente a más de un objeto particular, el polimorfismo es la capacidad de una operación para hacer frente a los objetos de más de un solo tipo.

Capítulo 7

Lenguajes Orientados a Objetos

La programación orientada a objetos hoy en día puede ser declarada un éxito. Los cursos introductorios de programación se suelen enseñar utilizando lenguajes orientados a objetos como C++ y Java. Los proyectos comerciales más nuevos eligen un lenguaje orientado a objetos. Sin embargo, las características orientadas a objetos fundamentalmente no añaden mucho a un lenguaje. No dan lugar a programas más cortos. El éxito de programación orientada a objetos se debe principalmente al hecho de que es un estilo que es adecuado para la psicología humana. Los seres humanos, como parte de su función básica, son expertos en reconocer e interactuar con objetos cotidianos. Además, los objetos en la programación son suficientemente parecidos a los objetos del mundo cotidiano que nuestra intuición se pueda aplicar a ellos.

Antes de discutir las propiedades de los objetos en programación orientada a objetos, vamos a revisar brevemente algunas de las propiedades más importantes de los objetos cotidianos que los hacen útiles para la programación.

- Los objetos cotidianos son activos, es decir, no están totalmente controlados por nosotros. Los objetos tienen un estado interno que está en evolución. Por ejemplo, un automóvil en marcha es un objeto activo y tiene un complejo estado interno, incluyendo la cantidad de gasolina restante, del refrigerante del motor y los niveles de transmisión de fluido, la cantidad

de energía de la batería, el nivel de aceite, la temperatura, el grado de desgaste de los componentes del motor, etc.

- Los objetos cotidianos son comunicativos. Podemos enviarles mensajes y podemos obtener información de objetos, como resultado del envío de mensajes. Por ejemplo, al encender un automóvil y comprobar el medidor de gas puede ambos ser vistos como el envío de mensajes al automóvil.
- Los objetos cotidianos están encapsulados. Tienen propiedades internas que no podemos ver, a pesar de que podemos aprender algunas de ellas mediante el envío de mensajes. Para continuar con el ejemplo del automóvil, comprobando el indicador de gasolina averiguamos cuánta gasolina queda, a pesar de que no podemos ver en el tanque de gas en forma directa y no sabemos si contiene gasolina extra o super.
- Los objetos cotidianos pueden ser anidados, es decir, los objetos pueden ser compuestos de varios componentes de objetos más pequeños, y esos componentes pueden, ellos mismos, tener componentes más pequeños. Una vez más, un automóvil es un ejemplo perfecto de un objeto anidado, al ser compuesto de objetos de menor tamaño como el motor y la transmisión. La transmisión, a su vez, es también un objeto anidado, incluyendo un cigüeñal, un convertidor de par y un conjunto de engranajes.
- Los objetos cotidianos poseen, en su mayor parte, un nombre único. Los automóviles tiene placas o números de chasis que los identifican de forma única.
- Los objetos cotidianos pueden ser conscientes de si mismos, en el sentido de que pueden intencionalmente interactuar con ellos mismos, por ejemplo, un perro que lame su propia pata.
- Las interacciones cotidianas de objetos pueden ser polimórficos en el sentido que un conjunto diverso de objetos pueden compartir un protocolo común de mensajería, por ejemplo, el sistema de mensajería acelerador-freno-volante de dirección es común para todos los modelos de automóviles y camiones.

Los objetos en programación orientada a objetos también tienen estas propiedades. Por esta razón, la programación orientada a objetos brinda un ambiente natural y familiar para la mayoría de la gente. Consideremos ahora los objetos de la variedad de programación.

- Los objetos tienen un estado interno en forma de variables de instancia o campos. Los objetos son por lo general activos y su estado no es totalmente controlado por sus interlocutores.
- Los objetos reciben mensajes, que son pedazos de código con un nombre, que están vinculados a un objeto en particular. De este modo, los objetos son comunicativos y grupos de objetos pueden realizar tareas mediante el envío de mensajes entre sí.
- Los objetos contienen código encapsulado (métodos u operaciones), junto con un estado mutable (variables de instancia o campos). Debe estar claro que los objetos son inherentemente no-funcionales. Esto quieren decir que guardan un estado. Esto refleja el estado de los objetos cotidianos.
- Los objetos se organizan normalmente en forma anidada. Por ejemplo, considere un objeto que representa un navegador web gráfico. El objeto en sí mismo es el marco, pero ese marco se compone de una barra de herramientas y un área de visualización. La barra se compone de botones, una barra de dirección y un menú, mientras que el área de visualización se compone de un panel y una barra de desplazamiento. La anidación de objetos se realiza generalmente cuando el objeto externo almacena sus objetos internos como campos o variables de instancia.
- Los objetos tienen nombres únicos, o referencias a objetos, para referirse a ellos. Esto es análogo a la denominación de los objetos del mundo real, con la ventaja de que las referencias a objetos son siempre únicas, mientras que en el mundo real los nombres de objeto puede ser ambiguos.
- Los objetos son conscientes de sí mismos, es decir, los objetos contienen referencias a los mismos. Estas auto-referencias se denomina this en Java y self en Smalltalk.
- Los objetos son polimórficos, es decir, un objeto "grueso" siempre se puede pasar a un método que toma uno "fino", es decir, uno con un menor número de métodos y campos públicos.

Hay varias características adicionales que los objetos comúnmente tienen. Las clases están casi siempre presentes en los lenguajes con objetos. Las clases no son necesarias: es perfectamente válido tener objetos sin clases. El lenguaje Self no tiene clases, sino que tiene objetos prototipo que se copian que hacen las veces de

clases. Conceptos importantes de las clases incluyen la creación, la herencia, la sobre-escritura de métodos, el acceso de la superclase y la distribución dinámica.

El ocultamiento de información para los campos y métodos es otra característica que la mayoría de los lenguajes orientados a objetos tienen, en general, en forma de palabras clave public, private y protected.

7.1. Historia

Contrariamente a la creencia de mucha gente, la Programación Orientada a Objetos (POO a partir de ahora) no es un tema nuevo de discusión. Es cierto que en años recientes la POO ha tomado nueva fuerza y ha renacido como un paradigma. Sin embargo, fue a finales de los años 60 cuando estas técnicas fueron concebidas. Para entender su origen hay que situarse en el contexto de la época, en el que el desarrollo y mantenimiento de proyectos software presentaban evidentes problemas de complejidad, coste y eficiencia. Uno de los grandes problemas que surgían consistía en la necesidad de adaptar el software a nuevos requisitos imposibles de haber sido planificados inicialmente. Este alto grado de planificación y previsión es contrario a la propia realidad. El hombre aprende y crea a través de la experimentación. La Orientación a Objetos brinda estos métodos de experimentación, no exige la planificación de un proyecto por completo antes de escribir la primera línea de código.

Esto mismo pensaron en 1967, Krinsten Nygaard y Ole-Johan Dahl de la Universidad de Oslo, en el Centro Noruego de Computación, donde se dedicaban a desarrollar sistemas informáticos que realizaban simulaciones de sistemas mecánicos, por ejemplo motores, para analizar su rendimiento. En este desarrollo se encontraban con dos dificultades, por un lado los programas eran muy complejos y, por otro, forzosamente tenían que ser modificados constantemente. Este segundo punto era especialmente problemático; ya que la razón de ser de los programas era el cambio y no sólo se requerían varias iteraciones para obtener un producto con el rendimiento deseado, sino que muchas veces se querían obtener diversas alternativas viables, cada una con sus ventajas e inconvenientes.

La solución que idearon fue diseñar el programa paralelamente al objeto físico. Es decir, si el objeto físico tenía un número x de componentes, el pro- grama también tendría x módulos, uno por cada pieza. Dividiendo el programa de esta manera, había una total correspondencia entre el sistema físico y el sistema informático. Así, cada pieza física tenía su abstracción informática en un módulo.

De la misma manera que los sistemas físicos se comunican enviándose señales, los módulos informáticos se comunicarían enviándose mensajes. Para llevar a la práctica estas ideas, crearon un lenguaje llamado Simula 67.

Este enfoque resolvió los dos problemas planteados. Primero, ofrecía una forma natural de dividir un programa muy complejo y, en segundo lugar, el mantenimiento pasaba a ser controlable. El primer punto es obvio. Al dividir el programa en unidades informáticas paralelas a las físicas, la descomposición es automática. El segundo punto también se resuelve. En cada iteración de simulación, el analista querrá cambiar o bien piezas enteras o bien el comportamiento de alguna pieza. En ambos casos la localización de los cambios está perfectamente clara y su alcance se reduce a un componente, siempre y cuando la interfaz del mismo no cambie. Por ejemplo, si se estuviese simulando el motor de un automóvil, puede que se quisiera modificar el distribuidor utilizado en la simulación anterior. Si el nuevo distribuidor tuviera la misma interfaz (mismas entradas y salidas) o se cambiase sólo su comportamiento interno, nada del sistema (a parte del distribuidor) estaría afectado por el cambio.

Con Simula 67 se introducen por primera vez los conceptos de clases, objetos, herencia, procedimientos virtuales y referencias a objetos (conceptos muy similares a los lenguajes Orientados a Objetos de hoy en día). En esta época, Algol 60 era el lenguaje de moda y Cobol el más extendido en aplicaciones empresariales, por lo que el nacimiento de Simula 67 y la Orientación a Objetos en Europa pasó inadvertido para gran parte de los programadores. En la actualidad Simula 67 se conoce simplemente como Simula, y contrariamente a lo que pudiera pensarse, todavía está en uso, mantenido por una pequeña comunidad de programadores, en su mayoría noruegos, y compiladores disponibles en la red.

El siguiente paso se da en los años 70 en los Estados Unidos, más concretamente en el Centro de Investigación de Palo Alto (PARC), California, donde Xerox tiene un centro de investigación en el que los científicos trabajan en conceptos que puedan convertirse en productos industriales al cabo de 10 a 20 años. En aquellos años contrataron a un joven llamado Alan Kay, que venía de la Universidad de Utah, donde había estado trabajando con gráficos, y había examinado un nuevo compilador procedente de Noruega, llamado Simula. Kay encontró conceptos, que más tarde aprovechó en Xerox, para llevar a término las ideas que proponía en su tesis doctoral. Éstas consistían básicamente en la propuesta de construcción de un computador llamado Dynabook, adecuado para ser utilizado por niños. El computador no tenía teclado, la pantalla era sensible al tacto y la mayor parte de la comunicación era gráfica. Al desarrollar este proyecto se

inventaron el mouse y los entornos gráficos. Al volver a encontrarse en Palo Alto con una programación compleja y experimental, como en el caso de Nygaard y Dahl, Kay, junto con Adele Goldberg y Daniel Ingalls, decidie- ron crear un lenguaje llamado Smalltalk. Este lenguaje de programación es considerado el primer lenguaje Orientado a Objetos puro, dónde todo lo que se crea son clases y objetos, incluyendo las variables de tipos más simples. La primera versión se conoció como Smalltalk-72, aunque fueron surgiendo nuevas versiones (76,80,..) que pasaron de ser desarrolladas sólo para máquinas Xerox a serlo para la mayoría de plataformas disponibles.

Hasta este momento, uno de los defectos más graves de la programación era que las variables eran visibles desde cualquier parte del código y podían ser modificadas incluyendo la posibilidad de cambiar su contenido (no existen niveles de usuarios o de seguridad, o lo que se conoce como visibilidad). D. Parnas fue quien propuso la disciplina de ocultar la información. Su idea era encapsular cada una de las variables globales de la aplicación en un sólo módulo junto con sus operaciones asociadas, de forma que sólo se podía tener acceso a estas variables a través de sus operaciones asociadas. El resto de los módulos (objetos) podían acceder a las variables sólo de forma indirecta mediante las operaciones diseñadas para tal efecto.

En los años ochenta surgen nuevos lenguajes orientados a objetos basados en Lisp. Esto se debe principalmente a que este lenguaje disponía de mecanismos que permitían la implementación basándose en orientación a objetos. Entre los lenguajes surgidos de Lisp podemos destacar Flavors, desarrollado en el MIT, Ceyx, desarrollado en el INRIA, y Loops, desarrollado por Xerox. Este último introdujo el concepto de la programación orientada a datos con la que se puede enlazar un elemento de datos con una rutina y que dio paso a la programación controlada por eventos. La mayoría de las vertientes tomadas por estos lenguajes se fusionaron en CLOS (Common Lisp Object System) que fue el primer lenguaje orientado a objetos que tuvo un estándar ANSI. En 1984, Bjarne Stroustrup de ATT-Bell se planteó crear un sucesor al lenguaje C, e incorporó las principales ideas de Smalltalk y de Simula 67, creando el lenguaje C++, quizás el lenguaje responsable de la gran extensión de los conceptos de la orientación a objetos. Posteriores mejoras en herramientas y lanzamientos comerciales de C++ por distintos fabricantes, justificaron la mayor atención hacia la programación Orientada a Objetos en la comunidad de desarrollo de software. El desarrollo técnico del hardware y su disminución del costo fue el detonante final. En esta misma década se desarrollaron otros lenguajes Orientados a Objetos como Objective C, Object Pascal, Ada y otros.

En el inicio de los años 90 se consolida la Orientación a Objetos como una de las mejores maneras para resolver problemas de programación. Aumenta la necesidad de generar prototipos más rápidamente (concepto RAD Rapid Aplication Development) sin esperar a que los requisitos iniciales estén totalmente precisos. En esta década se implanta con fuerza el concepto de reutilización, cuyo sentido es facilitar la adaptación de objetos ya creados a otros usos diferentes a los originales sin necesidad de modificar el código ya existente. A medio y largo plazo, el beneficio que puede obtenerse derivado de la reutilización es muy importante, y, quizás, es la razón principal por la que la industria informática se ha abocado a la orientación a objetos. Avanzando algunas cifras, se puede indicar que los niveles de reutilización de software pasan del 5-15 % en centros no orientados a objetos, a niveles por encima del 80 % en los orientados a objetos.

Con la reutilización como pilar básico de su filosofía, un equipo de Sun Microsystems crea Java en 1995, con el objetivo de conquistar el mundo de la programación, teniendo gran éxito en aplicaciones en red. No en vano, captó mucha atención en los primeros meses de 1996 en base a ser considerado el medio para dominar Internet. Java es totalmente orientado a objetos. Está basado en C++, intentando evitar los principales problemas del mismo, como, por ejemplo, el acceso directo a memoria dinámica. Se fundamenta en el uso de bytecode (código de bajo nivel, portable e interpretable) y una máquina virtual accesible para todo el mundo que ejecuta esos bytecodes. Hoy en día es lo más cercano a lo que podría denominarse una máquina universal.

El mundo de los lenguajes de programación orientados a objeto evoluciona día a día y, continuamente, surgen nuevos lenguajes o nuevas versiones de lenguajes ya consolidados.

7.2. Concepto de Clase y Objeto

El mundo está lleno de objetos: el automóvil, la lavadora, la mesa, el teléfono, etc. El paradigma de programación orientada a objetos proporciona las abstracciones necesarias para poder desarrollar sistemas de software de una forma más cercana a esta percepción del mundo real.

Mediante la POO, a la hora de tratar un problema, podemos descomponerlo en subgrupos de partes relacionadas. Estos subgrupos pueden traducirse en unidades autocontenidas llamadas objetos. Antes de la creación de un objeto, se debe definir en primer lugar su formato general, su plantilla, que recibe el nombre de

clase. Por ejemplo, pensemos que estamos intentando construir una aplicación de dibujo de cuadrados. Nuestro programa va a ser capaz de pintar cuadrados con diferentes tamaños y colores: uno será rojo y de 3 centímetros de lado, otro verde y de 5 centímetros de lado, etc. Como podemos deducir, cada cuadrado está definido por dos características, lado y color, y contiene la operación dibuja. Desde el punto de vista de la POO, cada cuadrado se va a representar como un objeto de la clase Cuadrado que es la contiene las características y operaciones comunes de los objetos. En realidad, estamos creando un nuevo tipo, el tipo Cuadrado, y cada variable de este tipo especial recibe el nombre de objeto.

Una clase, por lo tanto, puede definirse como un tipo de datos cuyas variables son objetos o instancias de una clase. Puede decirse que el concepto de clase es estático y el concepto de objeto es dinámico; ya que, sólo existe en tiempo de ejecución. Cada objeto de una clase comparte con el resto las operaciones, y a la vez posee sus propios valores para los atributos que posee: esto es, su estado. Las operaciones, también llamadas mensajes, se usarán tanto para cambiar el estado de un objeto como para comunicar objetos entre sí mediante el paso de mensajes.

Un ejemplo de clase podría ser la clase Alumno, esta clase definiría diferentes alumnos, en términos de nombre, número de matrícula, curso, especialidad, localidad, etc... el alumno Juan Sánchez con número de matrícula 5566432 sería una instancia de la clase Alumno. Otro ejemplo de clase podría ser la clase Teléfono, esta clase definiría diferentes teléfonos con los datos número, tipo de tarifa asociada, nombre de usuario asociado, e importe acumulado por cada teléfono. Una instancia de esta clase podría ser el teléfono con número 927123456.

7.3. Encapsulación

Una de los pilares básicos de la POO es la encapsulación de los datos. Según los principios de este paradigma de programación, el acceso a los datos de una clase debe realizarse de forma controlada, protegiéndolos de accesos no deseados. Cuando se desarrolla una aplicación, a veces es necesario ocultar los tipos de datos usados para que el usuario permanezca independiente de los detalles de los mismos. De esta manera, el usuario no es sensible a los cambios que se puedan producir en los tipos de datos elegidos dentro de una clase.

La encapsulación en los lenguajes orientados a objetos suele lograrse al declarar algunos datos como privados. El acceso a estos datos sería controlado; ya que, se

haría siempre a través de funciones miembro que realizarían las modificaciones oportunas de una forma transparente al usuario.

La encapsulación es el mecanismo que enlaza el código y los datos, a la vez que los asegura frente a accesos no deseados. La principal razón del uso de la encapsulación es evitar el acceso directo a atributos de una clase desde fuera de la propia clase. Se busca, principalmente, que el acceso a estos atributos se realice siempre mediante funciones miembro de la propia clase. El acceso a ciertos elementos de datos se puede controlar de forma estricta considerándolos privados.

Una vez presentado el concepto encapsulación, se puede definir de nuevo un objeto como una entidad lógica que encapsula los datos y el código que manipula dichos datos. Dentro de un objeto, parte del código y/o los datos puede ser privada al objeto y, por tanto, inaccesible desde fuera del mismo. Se podrá asegurar un acceso correcto al objeto utilizando las funciones miembro del objeto para acceder a los datos privados. El acceso a los datos de una clase se establece por lo tanto a través de los métodos que implementa la propia clase y que pone a disposición del usuario. Este mecanismo recibe el nombre de paso de mensajes o interfaz y es la forma de comunicación que se establece entre objetos.

7.4. Creación y Eliminación de Objetos

La creación y eliminación de objetos se realiza de la misma manera que para el resto de las variables de un programa. Normalmente, se crea cuando se declara y se destruye cuando termina su ámbito. Por ejemplo, si se declara un objeto teléfono dentro de una función, se comportará como una variable local, es decir, se creará cada vez que la ejecución del programa entre en dicha función y se destruirá cuando se llegue al final de la misma.

La única peculiaridad de los objetos consiste en que cuando se crean se invoca siempre a una función especial denominada constructor, mientras que cuando se destruyen se invoca siempre a una función especial denominada destructor. Éste suele ser el comportamiento general de los lenguajes orientados a objeto.

7.4.1. Constructores

Normalmente necesitamos inicializar alguno de los atributos miembro de un objeto, o todos, cuando se declara el objeto. Para realizar la inicialización de

los atributos, al definir una clase se utiliza un tipo especial de función miembro, ésta se llama constructor. Cuando se crea un objeto de una clase, se ejecutará automáticamente la función miembro constructor. El constructor se utilizará por lo tanto para inicializar los atributos miembro que se deseen de un objeto y para realizar algún otro tipo de inicialización que el usuario crea necesaria.

7.4.2. Destructores

El complemento de un constructor es el destructor. Al igual que a veces es necesario realizar inicializaciones u otro tipo de acciones al crear un objeto, a veces necesitamos realizar ciertas acciones cuando el objeto se destruye. Una de las ocasiones en las que se hace necesario el uso de un destructor es cuando se necesita liberar los recursos previamente reservados.

Algunos lenguajes orientados a objetos no definen explícitamente destructores para sus objetos, sino que utilizan otros mecanismos para la liberación de los recursos consumidos por los objetos. Un conocido mecanismo de esta índole es el recolector de basura, que se encarga de liberar los recursos de forma automática y transparente al programador. Este tópico se discute en el último capítulo de este libro.

7.5. Herencia

Desde el punto de vista de la programación orientada a objetos, la herencia permite definir una nueva clase extendiendo una clase ya existente. Esta nueva clase, clase derivada, posee todas las características de la clase más general y, además, la extiende en algún sentido. Mediante la herencia, una clase derivada hereda los atributos y métodos definidos por la clase base. De esta forma se evita volver a describir estos atributos y operaciones, obteniéndose así una importante reutilización de código.

Normalmente, las clases base suelen ser más generales que las clases derivadas. Esto es así porque a las clases derivadas se les suelen ir añadiendo características, en definitiva variables y funciones que diferencian, concretan y particularizan. La clase derivada, por lo tanto, tiene la posibilidad de extender las características heredadas de la clase base. Y lo puede hacer mediante una de estas tres maneras básicas:

- añadiendo nuevos miembros: atributos y métodos,
- modificando la visibilidad de los atributos y métodos heredados,
- proporcionando una nueva implementación de las operaciones heredadas: redefinición.

De la misma forma que la composición conlleva una relación entre los objetos implicados, la herencia implica la existencia de una relación entre la clase derivada y su clase base. Por lo tanto, a la hora de utilizar el mecanismo de la herencia hay que asegurarse de que se establece tal relación semántica entre la nueva clase y la clase original.

7.5.1. Jerarquías de Clases

El mundo real está compuesto por un número inconcebible de clases de objetos como, por ejemplo, el automóvil, la bicicleta, la motocicleta, etc. Para reducir la complejidad que supone concebir tal cantidad de objetos, la mente humana utiliza un mecanismo básico de relación entre clases de objetos: la clasificación. Mediante la clasificación se pueden agrupar clases de objetos que comparten una esencia común. De esta manera se van creando grandes clasificaciones, por ejemplo, Vehículos sería una clase que agruparía a las anteriores, pero también podríamos introducir un mayor nivel de detalle al introducir dos nuevas clasificaciones: Vehículos de dos ruedas y Vehículos de cuatro ruedas.

El mundo de la programación puede entenderse en muchos casos como un reflejo del mundo real. En este caso, la POO imita el mecanismo de clasificación a través del concepto de herencia. De esta manera, relacionando clases mediante la herencia podemos construir jerarquías de clases, en las que una clase puede ser a la vez clase base y clase derivada. Es interesante tener en cuenta que según bajamos por la jerarquía vamos obteniendo clases cada vez más especializadas, mientras que según subimos son más genéricas. En el ejemplo anterior, podríamos construir una jerarquía de clases que imitase la clasificación de vehículos observada en el mundo real. Para la construcción de esta jerarquía caben dos aproximaciones básicas: generalización o especialización.

En el primer caso, partimos de una serie de objetos que contienen una raíz común, por ejemplo automóvil, Bicicleta, Avión y Cohete. Extraemos esa raíz, código común, para crear una clase base que la contenga y de la que hereden las clases de objetos iniciales. En este caso, detectamos dos raíces: Vehículo

de tierra, clase base de automóvil y Bicicleta, y Vehículo de aire, clase base de Avión y Cohete. De esta manera, ya tenemos dos pequeñas jerarquías, que son susceptibles de ser agrupadas en una sola; ya que, poseen una raíz común: Vehículo.

Esta misma jerarquía es posible obtenerla a través de la especialización. En este caso, el proceso es a la inversa. Partimos de una clase genérica, como es Vehículo, y vamos creando nuevas subclases que extienden sus características, Vehículo de tierra y Vehículo de aire. Evidentemente, podemos seguir extendiendo estas nuevas clases para obtener clases aun más especializadas, como automóvil o Avión.

En conclusión, las características comunes de las clases de la jerarquía pertenecerán a la clase base, por ejemplo una variable que indique su velocidad, la función de arrancar y la de frenar, etc. Mientras que las características particulares de alguna de ellas pertenecerán sólo a la subclase, por ejemplo el número de platos y piñones, que sólo tiene sentido para una bicicleta, o la función despegar que sólo se aplicaría a los vehículos de aire.

Para hablar de herencia y jerarquías de clase estamos creando clases de objetos que aparecen de manera tangible en el mundo real; sin embargo, este reflejo tangible en la realidad no es un requisito indispensable, como ya sabemos, ni para la creación de clases, ni para el uso de la herencia, ni para la formación de jerarquías.

En definitiva, la herencia permite mediante la clasificación jerárquica gestionar de forma sencilla la abstracción y, por lo tanto, la complejidad.

7.6. Sobrecarga y Redefinición

En lo referente a la sobrecarga, ya vimos en el capítulo anterior qué significaba y cómo se lograba, apuntando también que no se trata de un concepto de orientación a objetos, sino más bien de una característica del lenguaje. Resumiendo, esta característica nos permitía crear varias funciones con el mismo nombre, pero con diferentes listas de parámetros o tipos de retorno. Por su parte, la redefinición, concepto introducido en este capítulo, nos permite crear clases derivadas que extiendan el comportamiento de la clase base, cambiando la definición de las funciones heredadas. Por lo tanto, hemos visto cómo se comportan estos

dos mecanismos por separado; sin embargo, nos falta saber cómo se comportarán cuando aparezcan juntos en el código de un programa. Para explicar su comportamiento se presenta un ejemplo de clasificación de documentos.

```
class Documento{
public:
  int archivar(){
    cout<<"Documento archivado"<<endl;
    return 1;
  }
  int archivar(int estanteria){
    cout<< "Documento archivado en "<< estanteria<< endl;
    return 1;
  }
};

class Libro: public Documento{
public:
  //Redefinición de una función sobrecargada
  int archivar(){
    cout<< "Libro archivado"<<endl;
    return 1;
  }
};

class Articulo: public Documento{
public:
  //Nueva versión: cambio de parámetros
  int archivar(char *clave){
    cout<< "artículo archivado por "<< clave<<endl;
    return 1;
  }
};

class Informe: public Documento{
public:
  //Nueva versión: cambio tipo de retorno
  void archivar(){
    cout<<"Informe archivado" << endl;
```

```
    }
};
```

En este caso tenemos la clase base, Documento, y tres clases que derivan de ella: Libro, Artículo e Informe. La clase Documento define dos versiones de la función archivar, es decir, esta función aparece sobrecargada. Por su parte, la clase Libro redefine una de las versiones de la función archivar. Mientras que las otras dos clases proporcionan una nueva versión de esa función: la primera, variando el número de parámetros y, la segunda, el tipo de retorno de la función. Ahora, la cuestión consiste en saber qué versiones de la función archivar están disponibles para los objetos de cada una de las tres clases derivadas.

```
int main() {
    int x;
    Libro libro;
    x = libro.archivar(); //error: versión no disponible
    x = libro.archivar(6);
    Articulo articulo;
    x = articulo.archivar(); //error: versión no disponible
    x = articulo.archivar("A");
    Informe informe;
    x = informe.archivar(); //error: versión no disponible
    informe.archivar();
};
```

Como se puede comprobar en el ejemplo, debemos ser conscientes de que la mezcla de sobrecarga y redefinición conlleva un comportamiento característico. En el caso del objeto de la clase Libro, la redefinición de una de las versiones de la función archivar supone que la otra versión pase a ser inaccesible. Por otro lado, como muestran las clases Articulo e Informe, si la clase derivada define una nueva versión de la función archivar, las versiones definidas en la clase base se hacen inaccesibles.

7.7. Polimorfismo

El polimorfismo es el tercer pilar de la programación orientada a objetos. Supone un paso más en la separación entre la interfaz y la implementación. Per-

mite mejorar la organización del código y simplificar la programación. Además, aumenta las posibilidades de extensión y evolución de los programas.

Está directamente relacionado con las jerarquías de clase. Básicamente, nos va a permitir que unos objetos tomen el comportamiento de objetos que se encuentran más abajo en la jerarquía, aumentando enormemente la expresividad del lenguaje. En definitiva, se trata de la posibilidad de que la identificación del tipo de un objeto se haga en tiempo de ejecución en vez de en tiempo de compilación. De esta manera, se pueden, incluso, construir estructuras en las que cada uno de sus elementos es de un tipo diferente.

Consideremos, por ejemplo, que hemos creado una jerarquía de clases de instrumentos musicales. Todos los instrumentos tienen la función tocar, pero cada uno tocará de una manera diferente. Mediante el polimorfismo vamos a conseguir crear, por ejemplo, un arreglo de instrumentos. Además, podemos tratar a los diferentes instrumentos de la misma manera, sin tener que hacer distinción entre sus diferentes tipos; sin embargo, vamos a obtener un comportamiento diferente para cada uno de ellos. Si creamos, por ejemplo, un lazo que recorra este arreglo de instrumentos invocando a la función tocar, esta función se comportará de manera distinta para cada uno: no es lo mismo tocar un violín que tocar un piano.

Capítulo 8

Sentencias de Control

El orden en que las sentencias son ejecutadas en un programa debe reflejar el algoritmo que se esta implementando. En un programa las sentencias se ejecutan de manera secuencial a menos que, utilizando sentencias de control, se altere la ejecución secuencial. Las secuencias de control pueden estar categorizadas en cuatro grupos:

- Expresiones y asignaciones
- Sentencias
- Programación declarativa o funcional
- Subprogramas

8.1. Expresiones y Sentencias de Asignación

Las expresiones son el mecanismo fundamental para especificar los cálculos en un lenguaje de programación. FORTRAN fue desarrollado para evaluar formulas escritas como expresiones en el lenguaje del programa. La manera como se evalúa, de forma automática, una expresión fue uno de los primeros objetivos de los lenguajes; Previo al desarrollo de FORTRAN, se trabajo dos años en

un preprocesador para evaluar expresiones; sin embargo, la evaluación de expresiones era un proceso lento hasta que, utilizando una notación matemática, desarrollada en Polonia en 1920, se redujo, en 1960 en un sistema Burroughs B5000, el tiempo de evaluación de expresiones a un diez por ciento del tiempo original.

Para la evaluación de expresiones, es necesario definir los siguientes temas:

- Reglas de precedencia de los operadores
- Reglas de asociación de los operadores
- El orden de evaluación de los operandos

8.1.1. Precedencia

La precedencia de las operaciones está definida en el lenguaje de programación. Si se considera la siguiente expresión:

$$3 + 4 * 7$$

Si la evaluación es de izquierda a derecha (primero la suma) el resultado es 49 y si es de derecha a izquierda (primero la multiplicación) el resultado será 31. La precedencia de las operaciones influye en el resultado, en la matemática la jerarquía de las operaciones indica que la multiplicación es de mayor jerarquía que la suma por lo que la operación es 3 + (4 * 7); La gramática de Smalltak define que las operaciones binarias tienen igual jerarquía por lo que en este caso la operación es (3 + 4) * 7.

La precedencia de los operadores aritméticos puede variar dependiendo del lenguaje. Por ejemplo en orden jerárquico:

Ruby	C	Smalltalk
**	Unitario - ++ –	unitario
Unitario +, -	* / %	binario
*, /, %	+ -	Palabras claves
Binario +, -		

Cuadro 8.1: Precedencia en diversos lenguajes

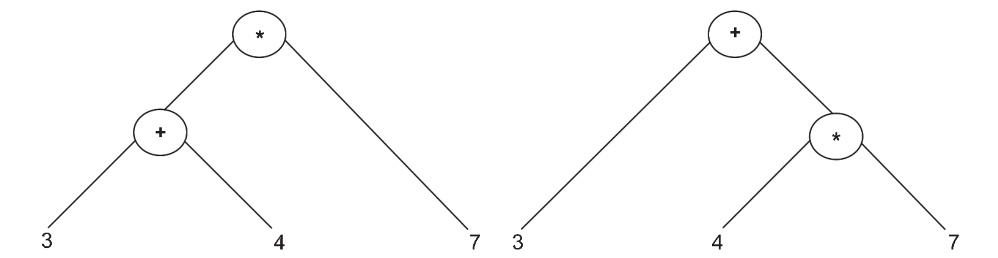


Figura 8.1: Árboles para la expresión 3 + 4 * 7. El primero evaluación de izquierda a derecha y el segundo evaluación de derecha a izquierda.

8.1.2. Asociación

Considere la siguiente expresión:

$$5 + 4 \uparrow 2 \uparrow 3 * 6 + 2$$

La asociación para los operadores de suma y multiplicación es de izquierda a derecha y para la exponenciación (↑) es de derecha a izquierda; por lo que en la expresión primero se evalúa $2 \uparrow 3$. La expresión, utilizando paréntesis para definir prioridades seria: $5 + ((\ 4 \uparrow (2 \uparrow 3))\ * 6) + 2$. El árbol que define la expresión es:

Las reglas de asociación son de izquierda a derecha excepto exponenciación que es de derecha a izquierda en la mayor parte de los lenguajes (en Visual Basic es de izquierda a derecha y en Ada no es asociativa).

Ruby tiene un orden de precedencia predefinido; sin embargo, al ser Ruby un lenguaje puramente orientado a objetos, los operadores son mensajes implementados en métodos que pueden ser redefinidos. En lenguajes funcionales como ML se puede redefinir la precedencia de los operadores.

8.1.3. Expresiones Aritméticas

La formula para encontrar el área de un polígono regular esta dada por:

(1/2) N sen(360°/N) S2

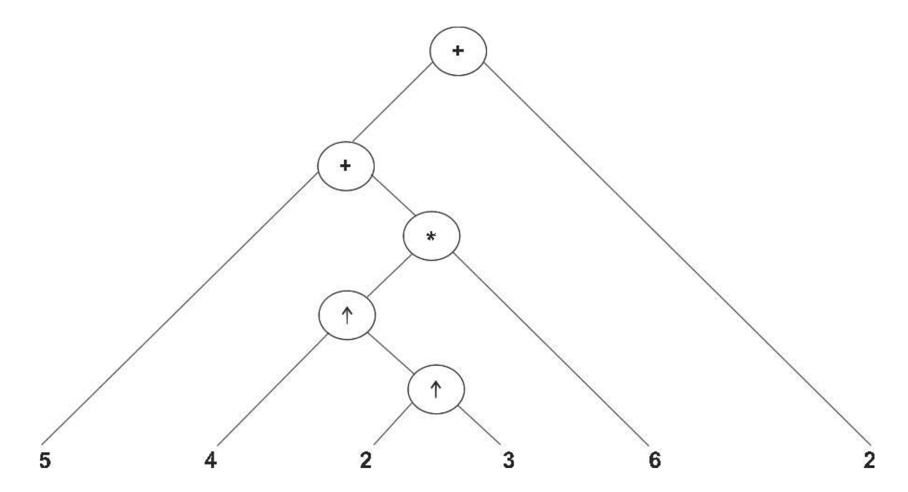


Figura 8.2: Árbol para la expresión $5 + 4 \uparrow 2 \uparrow 3 * 6 + 2$

La evaluación de esta expresión implica realizar algunas operaciones en las que hay que obtener los valores de las variables N y S. El método básico para evaluar expresiones es el de composición funcional que consiste en el proceso de combinar dos funciones o dos operandos en la que la una función u operando es evaluada primeramente y su valor remplazado en la otra. La composición funcional da lugar a que la evaluación pueda ser representada por un árbol en la que los nodos son los operadores y las hojas del árbol los operandos. La estructura del árbol lleva implícita la precedencia de las operaciones y la raíz del árbol es la operación principal. El árbol que representa la formula del área del polígono es:

El orden de evaluación de las operaciones, si primero se evalúa 1 / 2 o 360/N o S↑2 es decisión del que implementa el lenguaje.

La expresión esta escrita en notación infix (operando operador operando) con las reglas de precedencia comunes; la escritura de expresiones en notación infix necesitan paréntesis para definir precedencias adicionales a las ya establecidas. La estructura de árbol define este orden si recorremos el árbol de manera simétrica ("inorder") visitando:

1. El subárbol izquierdo
2. La raíz
3. El subárbol derecho

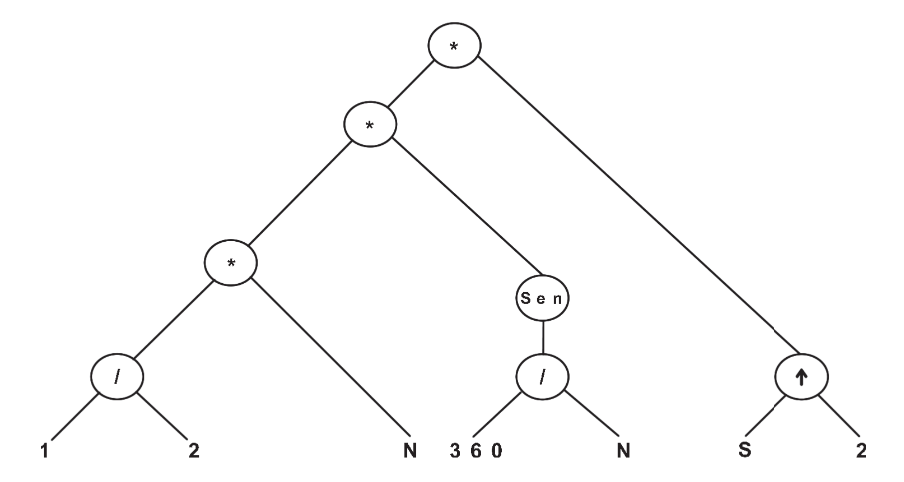


Figura 8.3: Árbol que representa la formula del área del polígono: (1/2)*N*sen(360°/N)*S↑2

8.1.4. Notación Polaca

Como se mencionó al inicio del capitulo, la evaluación de expresiones inicialmente no tenía una técnica eficiente hasta que se utilizo una forma de notación desarrollada por el matemático polaco Jan Lukasiewicz, a esta notación se la denominó notación polaca (operador operando operando) o prefix y la inversa (operando operando operador) se la conoce como notación polaca reversa (NPR) o postfix; ambas tienen la particularidad que no necesitan utilizar paréntesis y en NPR la evaluación de una expresión es sencilla.

La expresión en notación polaca como en notación polaca reversa pueden ser derivadas del árbol que representa la expresión.

Si recorremos el árbol en pre-orden visitando:

1. Raíz
2. Subárbol izquierdo
3. Subárbol derecho

Se tiene el siguiente resultado:

$$* * * / 1\ 2\ N\ \text{sen} / 360\ N \uparrow S\ 2$$

que es la expresión escrita en notación polaca.

Si se atraviesa el árbol en sentido de post-orden visitando:

1. Subárbol izquierdo
2. Subárbol derecho
3. Raíz

Se obtiene el siguiente recorrido:

$$1\ 2 / N * 360\ N / \text{sen} * S\ 2 \uparrow *$$

que es la notación polaca reversa de la expresión.

Nótese que en las notaciones infix, prefix y postfix el orden de los operandos en la expresión no se altera (1 2 N 360 N S 2).

8.1.5. Evaluación de una expresión postfix

Para la evaluación de una expresión usualmente se utiliza la notación postfix porque en la prefix primero se conocen los operadores y luego los operandos, lo que significa que hay que mirar hacia adelante para ejecutar la operación; la evaluación en notación postfix requiere de una pila temporal que almacene los operandos.

El algoritmo de evaluación es el siguiente: Se recorre la cadena de entrada;

si se encuentra

1. Un operando, se lo pone en la pila
2. Un operador de orden n, se ejecuta la operación sobre los n elementos que están en el tope de pila y se remplazan los n elementos en la pila por el resultado de la operación.

Si se utiliza la cadena:

$$1\ 2\ /\ N\ *\ 360\ N\ /\ \text{sen}\ *\ S\ 2\ \uparrow\ *$$

se tienen las siguiente operaciones:

1. Se pone 1 en la pila
2. Se pone 2 en la pila; gráfico (1)
3. Se ejecuta la operación 1 / 2
4. Se pone en la pila N; gráfico (2)
5. Se realiza la operación (1/2)*N y se pone el resultado en el tope de pila
6. Se pone 360 en la pila
7. Se pone N en la pila; gráfico (3)
8. Se realiza la división de 360 y N; gráfico (4)
9. Se aplica la operación unitaria seno sobre el tope de pila; gráfico (5)
10. Se pone en la pila S
11. Se pone en la pila 2; gráfico (6)
12. Se ejecuta S2; gráfico (7)
13. Se realiza la operación de multiplicación con el resultado en el tope de pila; gráfico (8)

8.1.6. Conversión de notación infix a postfix

La expresión en NPR puede ser obtenida a partir del árbol o se la puede obtener directamente de la expresión infix utilizando una pila en el proceso. El algoritmo es el siguiente:

La expresión en notación infix es la cadena de entrada (CE) y la expresión en notación postfix es la cadena de salida (CS). Se inicia el proceso barriendo la CE

1. Si el objeto es un operando se lo escribe en la cadena de salida CS

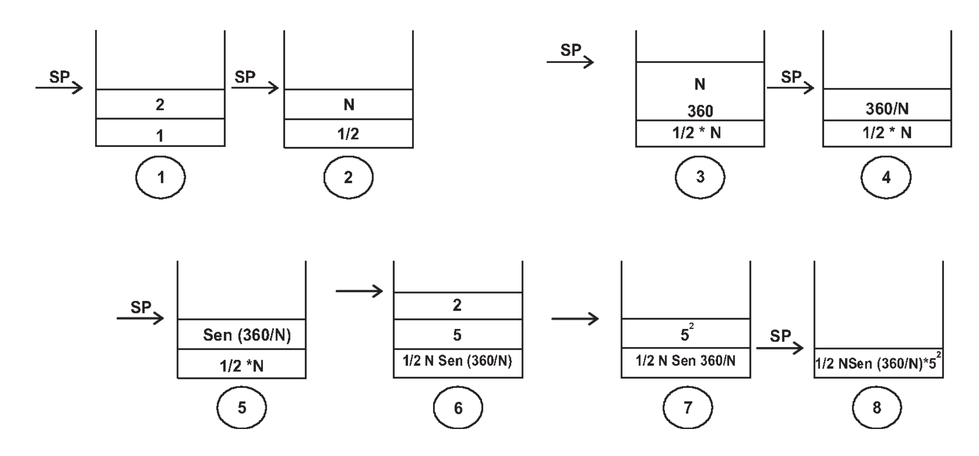


Figura 8.4: Estado de la pila con los operandos durante la evaluación de la expresión postfix 1 2 / N * 360 N / sen * S 2 ↑ *

2. Si es un operador se saca de la pila los operadores de igual o mayor precedencia, se los escribe en CS y se pone el nuevo operando en la pila

3. Si el objeto es un paréntesis izquierdo se lo pone en la pila

4. Si el objeto es un paréntesis derecho se saca de la pila todos los operadores hasta encontrar el paréntesis izquierdo y se los escribe en la CS

5. Al terminar la CE se sacan todos los operadores de la pila y se los escribe en la CS

Si se utiliza la expresión (1/2)*N*sen(360°/N)* S↑2 como cadena de entrada CE, aplicando el algoritmo se tiene:

1. Se pone "(" en la pila (regla 3)

2. Se pone 1 en la CS (regla 1), se lee se siguiente objeto "/" y se lo pone en la pila (regla 2)

3. Se pone 2 en la CS (regla 1), se lee el siguiente objeto ")" y se aplica la regla 4

4. Se pone "*" en la pila (regla 2)

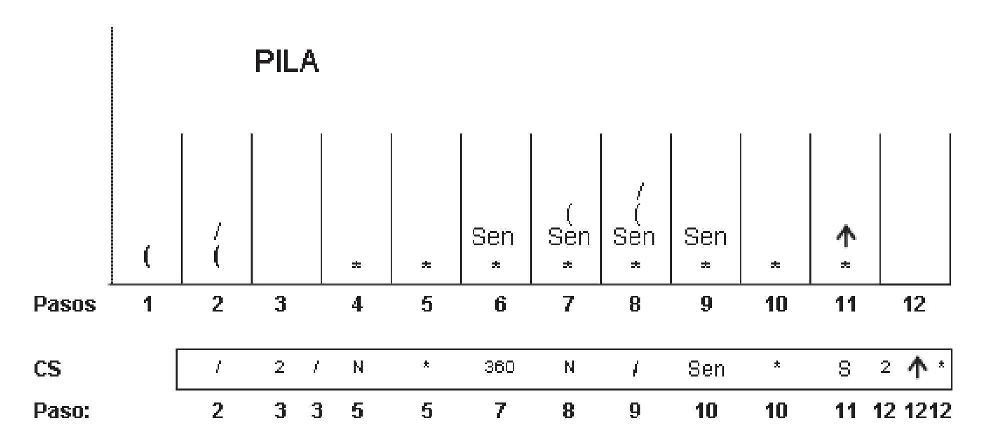


Figura 8.5: Estado de la pila y cadena de salida durante la ejecución del algoritmo

5. Se escribe N en la CS, se lee “*” de la CE y se aplica regla 2
6. Se lee “sen” y se lo pone en la pila (regla 2)
7. Se pone “(“en la pila (regla 3) y 360 en la CS (regla 1)
8. Se lee “/” y se lo pone en la pila (regla 2), se lee N y se lo pone el CS (regla1)
9. Se lee “)” y se aplica regla 4
10. Se lee “*” y se aplica regla 2
11. Se pone S en la CS, se lee “↑” y se aplica regla 2
12. Se lee 2 se lo pone en la CS, se termina la CE y se aplica la regla 5

8.1.7. Orden de evaluación de los operandos

La utilización de la notación infix, prefix y postfix puede conducir a una ambigüedad en lo que es un operando. El orden de evaluación depende, de la precedencia de los operadores, de la asociación (asociación por la izquierda o por la derecha); estos temas están definidos en la gramática del lenguaje. El

orden de evaluación de los operadores es importante cuando la evaluación de un operando tiene efectos colaterales.

Un subprograma tiene efectos colaterales cuando cambia de valor una variable externa al subprograma (una variable global); por ejemplo:

a-f(b) – c*d

Puede ejecutarse primero (a – f(b)) o (c*d); si el evaluar f(b) modifica a d como efecto colateral, entonces el valor de la expresión dependerá del orden de evaluación de los operandos. Este error no ocurre en lenguajes funcionales como ML ya que las funciones no tienen efectos colaterales debido al pase por valor y al ámbito del almacenamiento en memoria de los identificadores.

8.1.8. Reordenamiento de las expresiones

El arreglar expresiones puede conducir a sobrecarga aritmética; por ejemplo en la expresión a-b+c-d; si se asume que a, b, c y d son números grandes y se arregla la expresión como (a+c) – (b+d), uno de los resultados parciales puede dar un error por sobrecarga aritmética. Usualmente los lenguajes tienen implementado la detección de sobrecarga en sus operaciones.

8.1.9. Evaluación de cortocircuito

La evaluación de expresiones booleanas en cortocircuito significa que en el cálculo de la expresión, una vez que el resultado está determinado, ya no es necesario continuar la evaluación completa de la expresión lógica; por ejemplo, si la expresión es $A \bigwedge B$ y al evaluar A su valor es verdadero ya no es necesario la evaluación de B. La evaluación completa de la expresión se la denomina "evaluación ansiosa" (eager evaluation). Los lenguajes usualmente tiene los dos tipos de evaluación, Pascal es uno de los lenguajes que no posee evaluación de cortocircuito.

8.2. Control de Secuencias entre Sentencias

Los mecanismos básicos para controlar el flujo de las operaciones en un programa son considerados a continuación.

8.2.1. Asignaciones

Los cambios de estado principales se producen al asignar el valor de una expresión a una variable. En la asignación A := B, el r-value de B es asignado a la dirección dada en el l-value de A.

La sintaxis varía dependiendo de la gramática del lenguaje:

Lenguaje	Asignación
Ada	A := B
C, Fortran, ML	A=B
Cobol	MOVE B TO A
APL	A <- B
LISP	(SETQ A B)

Existen diferencias fundamentales entre las asignaciones de los lenguajes imperativos y funcionales.

- En los lenguajes imperativos: se calcula por medio de efectos colaterales; la computación es una serie de cambios a los valores de las variables en memoria.
- En los lenguajes funcionales los valores dependen del ámbito referencial de la expresión y no de cuando se la evalúa. La evaluación siempre retorna los mismos resultados dados los mismos argumentos porque no tiene efectos colaterales.

Combinaciones de operadores de asignación:

Lenguaje	Asignaciones
C/C++	a+=b, a-=b, ++A, –A, A++, A—
ML, Perl, Ruby	a,b:=c,d Intercambio a,b :=b,a a,b :=1

8.2.2. Flujo Estructurado y no Estructurado

Se definen los flujos no estructurados a las sentencias goto. El origen del goto proviene de los lenguajes de máquina de los procesadores en los que existen

instrucciones de salto. FORTRAN adoptó esta técnica y la sentencia goto transfería el control de la secuencia a una dirección dada por una etiqueta. Si bien el proceso es sencillo ha sido muy cuestionado, E. Dijkstra en 1968 planteo la eliminación del uso del goto por ser mala práctica de programación, se demostró que se puede programar sin la construcción goto y se promovió la metodología de programación estructurada.

Existen dos clases de saltos: el goto incondicional tal como

```
goto 200
```

que transfiere el control a la sentencia que tiene como etiqueta 200.

El goto condicional transfiere el control si se da la condición:

```
if A>B then goto 200
```

Si bien no es necesario el salto incondicional de goto, es importante la utilización de saltos predefinidos; se tiene en C la sentencia "break" que transfiere el control a la siguiente sentencia después de la malla donde se ejecuto la instrucción; otra construcción es el "continue" que salta al final de la malla para que continúen las iteraciones. Ruby es mas versátil, tiene break, next, redo y retry;

Los flujos estructurados son: las secuencias, sentencias condicionales, iteraciones y subrutinas.

Secuencias es una lista de sentencias que son ejecutadas secuencialmente. Una sentencia compuesta es una secuencia de sentencias que pueden ser tratadas como una sola sentencia; una sentencia compuesta es un bloque si incluye declaración de variables, C, C++, Perl, Java, Ruby utilizan "{" y "}" para delimitar el bloque. Pascal, Modula, Ruby utilizan begin . . . end como delimitadores.

Sentencias condicionales expresan la alternativa entre dos o más sentencias; en la sintaxis de C:

```
if (<expr>) <sentencia> [else <sentencia>]
```

si la expresión es verdadera se ejecuta la primera sentencia sino la segunda; en C la expresión puede ser de valor entero, en Java debe ser de tipo booleano.

La sintaxis de Ada permite múltiples condiciones:

```
if <cond> then
  <sentencias>
```

```
elseif <cond> then
  <sentencias>
...
else
    <sentencias>
end if
```

Sentencias case/switch que se utilizan como remplazo de los if anidados. En Ruby la sintaxis es:

```
case
when <condición> [, <condición>] ...[then | :]
  <sentencias>
when <condición> [, <condición>] ...[then | :]
  <sentencias>
...
[else
  <sentencias>]
end
```

o

```
case <objetivo>
when <comparación> [, <comparación> ... [then | :]
  <sentencias>
when <comparación> [, <comparación> ... [then | :]
  <sentencias>
[else
  <Sentencias>]
end
```

La implementación del case es sencilla; en memoria una tabla de saltos en la que esta la dirección donde se encuentra el bloque de sentencias que se va a ejecutar.

Iteraciones son el mecanismo básico para cálculos repetidos de una colección de sentencias. El control del número de iteraciones puede darse por un contador que contabiliza el número de repeticiones o por una lógica booleana en la que se repite la iteración hasta que la variable booleana cambia de estado; la prueba de la condición puede ser realizada al inicio o final de la iteración o internamente con una condición de salida. Uno de los lenguajes más ricos en instrucciones de control iterativo es Ruby:

```
while <condición> do
...
end

for <condición> do
...
end

loop do
...
break <condición>
...
end
```

```
until <condición> do
...
end

lista.each do
...
end

n.times do
...
end
```

Las sentencias de control son de diferentes clases; saltos incondicionales, saltos definidos, secuencias, sentencias condicionales, case/switch, iteraciones y subprogramas que se estudiará en el siguiente capitulo.

Existe un número diverso de maneras de expresar una malla y la mayor parte de los lenguajes tienen mecanismos para interrumpir la ejecución normal de la malla. Ruby tiene iteradores que son métodos enviados a varios objetos y además permite que el programador defina iteradores.

Capítulo 9

Control de Subprogramas

En lenguajes, un subprograma (llamado también: procedimiento, subrutina, rutina, función, o método) esta definido como una porción de código dentro de un programa que realiza una tarea específica y es relativamente independiente del código restante.

El subprograma se comporta de la misma forma que un programa que es utilizado en un programa más grande. Durante la ejecución de un programa, un subprograma puede ser llamado varias veces y de diferentes lugares.

El concepto se desarrolló inicialmente en lenguajes ensambladores con el nombre de macros, introducido por Maurice Wilkes en 1951, que involucra secuencias de líneas de texto en los que están insertados variables y constantes. Una vez que el macro ha sido definido, su nombre puede ser utilizado como un mnemónico. Cuando el ensamblador procesa esa sentencia, remplaza la sentencia con las líneas de texto asociadas con el macro y las procesa como si existieran en el código fuente. Los macros pueden tomar parámetros que son remplazados en el momento de la llamada.

El concepto de macro fue utilizado en FORTRAN, la dirección de retorno del macro era almacenada en una posición fija de memoria adyacente al código de la subrutina lo que no permite que el macro sea llamado de nuevo mientras la llamada anterior esta activa; este tipo de implementación impedía utilizar el concepto de recursión, en FORTRAN 90 la recursividad se estandarizó.

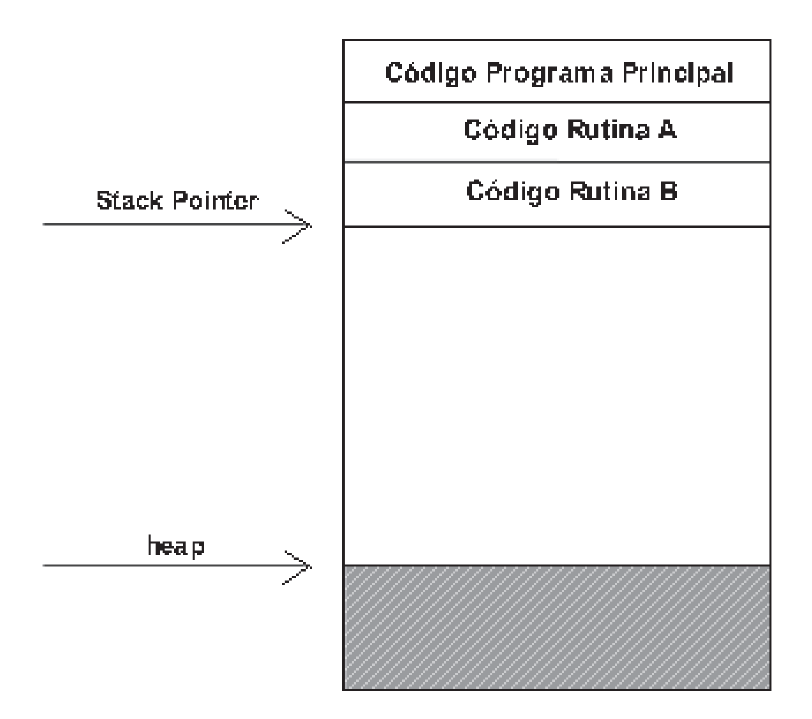


Figura 9.1: Generación de llamadas de un solo flujo

En la Figura 9.1, el programa principal llama a la rutina A de la que retorna cuando se ejecuta la instrucción return; la rutina A llama a la rutina B de la que retorna con la instrucción de return de la rutina, la rutina B es llamada dos veces. Estas ejecuciones simples son call-return.

9.1. Registros de Activación

Para que un lenguaje pueda utilizar recursión, es necesario que en cada llamada se almacene la dirección de retorno y las variables y parámetros que son utilizados durante la ejecución del subprograma.

El subprograma tiene un segmento de código que es el código ejecutable y las constantes, este segmento de código es invariante durante la ejecución del programa y se compila y almacena junto con el programa principal.

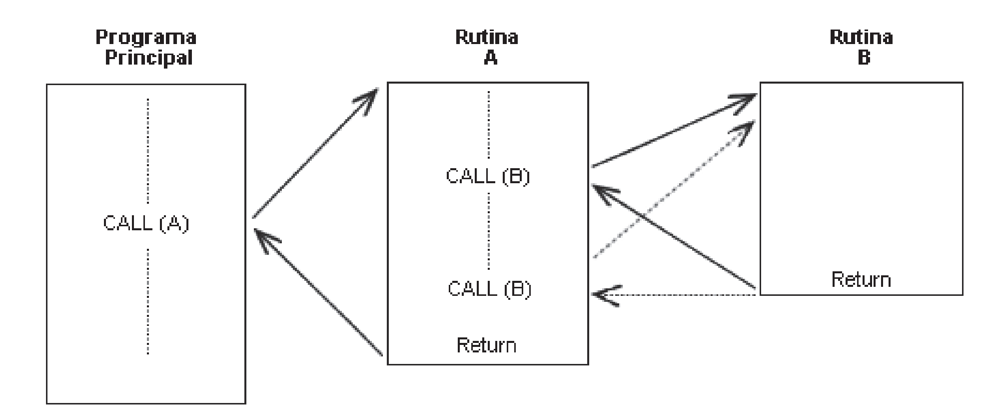


Figura 9.2: Modelo de la memoria después de compilar el programa y las dos rutinas A y B

Cuando el subprograma es llamado, se crea dinámicamente un registro de activación que contiene las variables locales, los parámetros de entrada, los valores de salida y la dirección de retorno al programa de donde fue llamado el subprograma. Cada vez que se llama al subprograma se crea un registro de activación en la memoria apuntada por el puntero de pila (stack pointer) y el registro de activación se destruye cuando se retorna del subprograma.

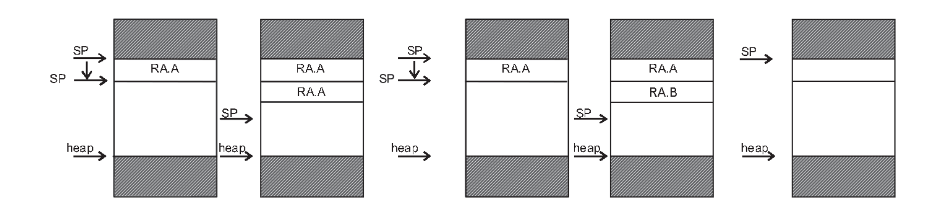


Figura 9.3: Memoria durante las Fases de la ejecución de las llamadas del programa de la Figura 9

En el modelo de memoria que se utiliza en el texto, se tiene que el código del programa, las variables estáticas (que están ligadas a una posición específica de memoria hasta que el programa termina de ejecutarse) y las subrutinas junto

con las constantes de la subrutina, están cargadas en lo que se conoce como la memoria estática y el puntero de pila apunta al inicio de la ejecución del programa a esa dirección; durante la ejecución del programa cada llamada a una subrutina crea un registro de activación y el puntero de pila se incrementa, cuando se ejecuta "return" el registro de activación se destruye junto con las variables locales y el puntero de pila decrece. La creación de variables dinámicas (creadas por malloc, new) durante la ejecución del programa son asignadas en el heap que se va incrementando. El incremento de la pila y del heap reduce cada vez más el espacio disponible; cuando el espacio se torna crítico es necesario optimizar el heap (desfragmentar y compactar).

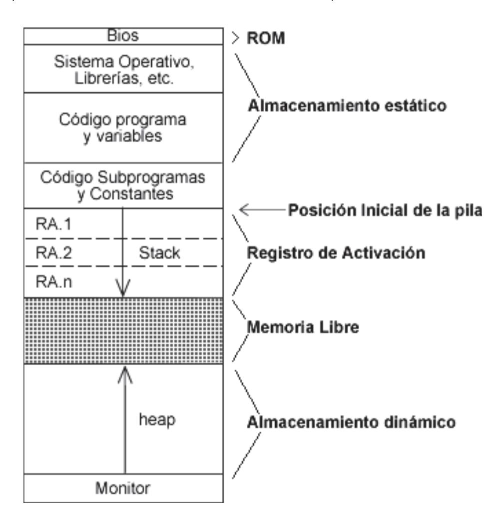


Figura 9.4: Organización de la memoria durante la ejecución de subprogramas

Considere el siguiente subprograma en C:

```
int ejem_reg  (int a, float b, char c) {
  const valinic = 10;
   int i, j, k;
  double m;
  ...
  ...
  return (j+k+5); }
```

En la compilación del subprograma se almacenará el código y las constantes (10,

5) y, en el registro de activación las variables, parámetros de entrada y salida y dirección de retorno.

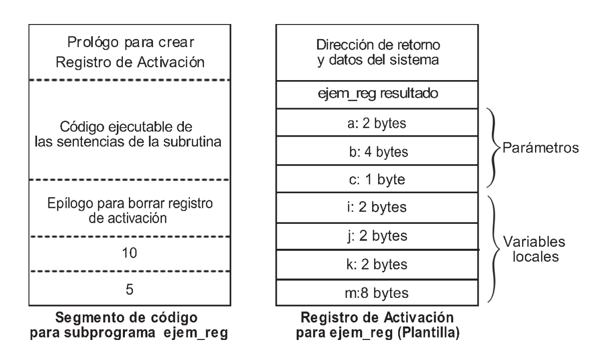


Figura 9.5: Memoria de almacenamiento del código y registro de activación generado al llamar a ejem_reg

Nótese que existe un código con instrucciones de cómo crear el registro de activación (Prologo) así como un epilogo para borrar el registro de activación y desaparecer todas las variables locales.

9.2. Corutinas

Las subrutinas son un caso especial de las corutinas; cuándo se llama a una subrutina, la ejecución comienza al inicio y una vez que encuentra un "return" se termina la subrutina. Las corutinas son similares, excepto que salen de la corutina regresando desde donde fue llamada o llamando a otra corutina.

En la Figura 9.6 se ejecutan dos corutinas en paralelo, en un esquema similar a los hilos. Las subrutinas son implementadas en una sola pila, las corutinas tienen una arquitectura diferente en la que se utilizan pilas adicionales bajo el concepto de continuaciones. Lenguajes ensambladores y lenguajes de alto nivel como Ruby, Lua, Go soportan corutinas.

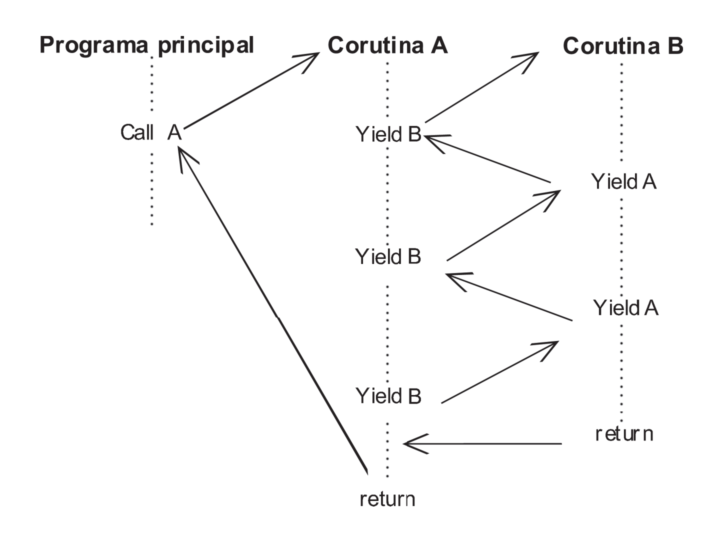


Figura 9.6: Ejecución de dos corutinas A y B

9.3. Atributos de Control de Datos

Los datos en un programa serán utilizados desde diferentes partes del programa que se encuentra en ejecución. El control de datos especifica, en un lenguaje, la forma como acceder al dato para cada operación y como el dato resultante de una operación es almacenado para su uso posterior. Se puede escribir el método:

```
void m (int i) {
   i = 4;
}
```

en el que i es un identificador al que se le asigna un valor que puede cambiar por lo que es una variable; al objeto de dato se le ha dado un nombre cuando fue creado. Un ejemplo más interesante es el patrón utilizado en programación orientada a objetos; por ejemplo, en Ruby:

```
def nombre
   @un_nombre
end
def nombre= (y)
   @un_nombre = y
```

```
end
```

la variable de instancia un_nombre es privada (en la definición de Ruby) y solo se puede acceder por el método nombre (getter) y modificar a través del método nombre= (setter).

El conjunto de asociaciones o uniones (binding) que se generan al activarse un subprograma es denominado entorno referencial. Este entorno puede ser:

- Local. El conjunto de asociaciones creadas a la entrada de un subprograma que pueden ser parámetros formales, variables locales o nombres de subprogramas definidos dentro del subprograma.
- No locales. El conjunto de asociaciones para identificadores usados dentro del subprograma o que no han sido creados a la entrada del mismo. Las asociaciones predefinidas o las creadas al inicio de ejecución del programa (globales) y que son usadas en el subprograma son parte del entorno referencial.

9.3.1. Alias para Objetos de Datos

Durante la ejecución de un programa un objeto de datos, que se encuentra en una posición de memoria, puede tener accesos con diferentes nombres en el programa. Una manera común de generar alias es el de referenciar una posición de memoria con más de un nombre (por ejemplo, con punteros). Al modificar el objeto a través de un nombre, implícitamente modifica el valor asociado a los otros nombres o alias pudiendo producirse resultados inesperados para el programador; por ejemplo, la función en C que implementa un algoritmo para intercambiar datos utilizando la operación lógica XOR:

```c
void xorSwap (int *x, int *y) {
     if (x != y) {
          *x ^= *y;
          *y ^= *x;
          *x ^= *y;
     }
}
```

El algoritmo funciona si los punteros x y y son distintos; si fueran iguales (alias el uno del otro) la función fallaría (el resultado sería cero). A más de generar código difícil de entender, uno de los conflictos con la utilización de alias se da durante la compilación en la fase de optimización de código; por tal motivo, el estándar ISO para C especifica que es ilegal, para punteros de diferentes tipos, referenciar la misma posición de memoria. Esta regla es conocida como alias estricto.

9.4. Ambiente Estático y Dinámico

El ámbito (scope) es el contexto dentro de un programa en el cual el nombre de una variable u otro identificador es válido y puede ser utilizado. Fuera del ámbito de una variable, el valor de variable puede todavía ser almacenado y puede aún ser accesible de alguna forma pero el nombre no se refiere a ese valor; esto es, el nombre no está ligado al almacenamiento de la variable.

9.4.1. Ámbito de la Función

Los lenguajes de programación crean variables locales en un subprograma. Estas variables locales desaparecen cuando se regresa del subprograma (el registro de activación es destruido).

Por ejemplo, en el siguiente código en JavaScript:

```
function cuadrado(n) {
  return n * n;
}
function suma_de_cuadrados(n) {
  var ret = 0;  // el valor a retornar
  var i = 1;    // un contador de 1 a n
  while(i <= n) {
    ret = ret + cuadrado(i);
    i = i + 1;
  }
  return ret;
}
```

cada una de las dos funciones tiene un argumento con nombre "n"; las dos variables "n" no están relacionadas, cada una es una variable local con un ámbito en su función.

9.4.1.1. Ámbito de Bloque en una Función

Algunos lenguajes permiten que las variables sean visibles en una parte del subprograma y que tengan un ámbito de bloque. En el siguiente código en C:

```
int suma_de_cuadrados(int n) {
  int ret = 0;
  int i = 1;
  while(i <= n) {
    int n = i * i;
    ret = ret + n;
    i = i + 1;
  }
  return ret;
}
```

Se tiene dos variables n, dentro de la malla while (i<=n) se crea, cada vez que se ejecuta la malla, una variable n que es válida solamente dentro de la malla. La variable externa n no es visible en la malla excepto en el encabezamiento de la misma (while(i<=n)).

Lenguajes como C# y Java soportan ámbito de bloques pero no permiten que una variable local esconda otra. En esos lenguajes, utilizar la segunda n genera un error sintáctico.

9.4.1.2. Expresiones let

Lenguajes funcionales como ML tienen expresiones let que permiten que el ámbito de una declaración sea una sola expresión.

Esto es conveniente cuando se requieren valores intermedios para un cálculo. Por ejemplo, en ML, si f() retorna 12 entonces:

```
let val x = f() in x * x end
```

es una expresión que da un valor de 144 utilizando una variable temporal llamada x para evitar llamar a f() dos veces.

9.4.1.3. Ámbito Global

Una declaración tiene ámbito global si tiene efecto en todo el programa. Los nombres de variables con ámbito global se las denominan variables globales y se considera mala práctica el utilizarlas por el programador; el lenguaje las utiliza para nombrar las clases, mensajes, funciones, etc. En los códigos anteriores cuadrado y suma_de_cuadrados tienen ámbito global.

Como se vio anteriormente el ámbito de función y el de bloque son útiles para evitar colisión de nombres y para ayudar a mitigar este problema algunos lenguajes utilizan "namespaces".

9.4.1.4. Ámbito Léxico y Dinámico

En el uso de variables locales dentro de una función es importante aclarar el concepto de "estar dentro de la función"; existen dos maneras diferentes de aclarar este concepto.

En el ámbito léxico usualmente llamado ámbito estático el ámbito de una variable definida en una función es el texto del programa que define la función; en cambio, en el ámbito dinámico, el ámbito de la variable que está en una función es el periodo de tiempo en el que la función se está ejecutando; mientras la función esta corriendo, el nombre de la variable existe. Si se considera el siguiente programa:

```
x=1
function g () { echo $x ; x=2 ; }
function f () { local x=3 ; g ; }
f #  imprime 1, o 3?
echo $x #  imprime 1, o 2?
```

La primera línea crea una variable global x que se inicializa en 1. La segunda línea defina una función g que imprime el valor actual de x y luego lo fija en 2. En la tercera línea f crea una variable local x y la inicializa con 3 y luego llama a g. La cuarta línea llama a f. La quinta línea imprime el valor actual de x.

¿Qué imprime el programa? Depende si el lenguaje utiliza ámbito estático o dinámico; si utiliza ámbito estático, entonces g imprime y modifica x (porque g está definido fuera de f) por lo que el programa imprime 1 y luego 2. Si el lenguaje utiliza ámbito dinámico entonces g imprime y modifica la variable local

x en f (porque g es llamado dentro de f) entonces el programa imprime 3 y luego 1. (El lenguaje del programa es Bash y es de ámbito dinámico).

9.4.1.5. Ámbito Estático

Con el ámbito léxico o estático, un nombre siempre se refiere a su entorno local estático. Esta es una propiedad del texto del programa e independiente de las llamadas en tiempo de corrida; el análisis es por lo tanto estático. Ámbito estático es estándar en los lenguajes basados en Algol tales como Pascal, Modula2 y Ada y en lenguajes modernos funcionales como ML y Haskell. Es también utilizado en C y similares lenguajes.

Considere el siguiente ejemplo en Pascal:

```
program A;
var I:integer;
    K:char;

    procedure B;
    var K:real;
        L:integer;

        procedure C;
        var M:real;
        begin
          (*ambito A+B+C*)
        end;

      (*ambito A+B*)
    end;

  (*ambito  A*)
end.
```

En el programa A, ámbito de la variable I es global, la variable K, tipo caracter, solo es visible en A, por cuanto en P se define la variable K real, que es visible en B y C de igual forma que L. La variable real M es visible en C y no se puede acceder de B o A.

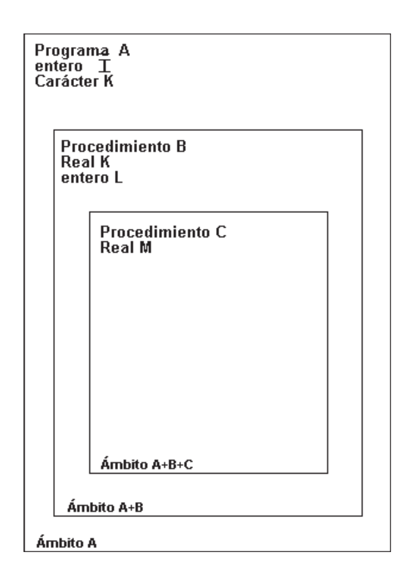


Figura 9.7: Ámbitos de programa A y procedimientos B y C

9.4.1.6. Ámbito Dinámico

Con el ámbito dinámico, cada identificador tiene una pila global de uniones o asociaciones. El introducir una variable local con nombre x pone una asociación en la pila de x que es sacada cuando le flujo de control abandona el ámbito. La asociación será con el entorno más reciente. Nótese que esto no puede ser realizado en tiempo de compilación sino en tiempo de corrida.

El ámbito dinámico es sencillo de implementar; para encontrar un valor es necesario recorrer los registros de activación por un valor para el identificador. En la práctica, se utiliza una lista de asociaciones que es una pila, tipo LIFO, de pares nombre/valor.

En el siguiente programa de ejemplo:

```
program principal;
   integer y;

   procedure M ( );
         write y;
```

```
    end;

    procedure N ( );
         integer y;
         y := 1;
         M ( );
   end;

begin
   y := 5;
   M ( );
   N ( );
end.
```

Si el lenguaje es de ámbito estático la ejecución daría los valores mostrados en el Cuadro 9.1. Si el ámbito es dinámico los valores resultantes difieren como se muestra en el Cuadro 9.2.

Variables y valores	*Notas*	Pila
principal.y = 5	En principal	principal
principal.y= 5	M es llamadoImprime y	principal | M
principal.y = 5	Retorna a principal	principal
principal.y = 5N.y=1	N es llamado	principal | N
principal.y=5N.y=1	M es llamadoImprime y	principal | N | M
principal.y=5N.y=1	Retorna a N	principal | N
principal.y=5	Retorna a principal	principal
Salida 5 5		

Cuadro 9.1: Ámbito Estático

El tiempo de vida de una variable es el periodo en el que la variable existe; si la variable esta declarada en un procedimiento, la variable existe desde que se crea el registro de activación; entonces el tiempo de vida de esa variable es el periodo en que existe el registro de activación.

El tiempo de vida de x en M esta dado por la duración del registro de activación y el ámbito se extiende hasta P donde se define la variable local x.

Variables y valores	*Notas*	Pila
principal.y = 5	En principal	principal
principal.y= 5	M es llamado desde principalImprime y	principal | M
principal.y = 5	Retorna a principal	principal
principal.y = 5N.y=1	N es llamado	principal | N
principal.y=5N.y=1	M es llamado desde NImprime y	principal | N | M
principal.y=5N.y=1	Retorna a N	principal | N
principal.y=5	Retorna a principal	principal
Salida 5 1		

Cuadro 9.2: Ámbito Dinámico

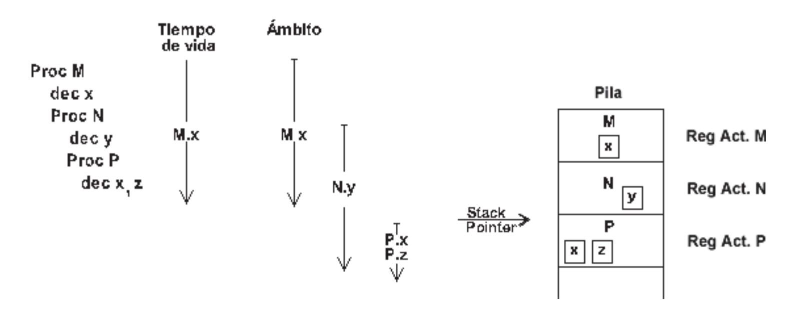


Figura 9.8: Tiempo de vida de las variables

9.5. Transmisión de Valores entre Subprogramas

El programador al llamar a un procedimiento lo hace usando parámetros actuales; estos parámetros actuales remplazan a los parámetros formales que se utilizan en la definición del subprograma. El subprograma se ejecuta usando los nombres de los parámetros formales para acceder los valores pasados como parámetros actuales; también puede retornar un resultado. Los argumentos y resultados se aplican a los datos enviados y retornados del subprograma a través de diversos mecanismos del lenguaje.

La declaración siguiente en Ada:

```
procedure desplazamiento (a: out integer, b: in out
    integer, c: in integer) is
begin
```

```
  a := b; b := c;
end desplazamiento;
```

utiliza 3 parámetros formales: "a" que es un parámetro de salida (resultado) y que se puede escribir pero no leer (debe estar en el lado izquierdo de una asignación); "b" que es un parámetro de entrada salida; y, "c" que es un parámetro de entrada y que solo se puede leer (debe estar enel lado derecho de una asignación).

9.5.1. Pase de Parámetros

Los parámetros mapean los parámetros actuales en el punto de llamada a los parámetros formales en el subprograma, Esto permite que el programador pueda escribir un procedimiento sin conocer el contexto en que el procedimiento va a ser ejecutado y permite llamar al procedimiento con diversos contextos sin conocer la operación interna del procedimiento. Los parámetros formales son remplazados por los parámetros actuales usualmente por correspondencia posicional, el remplazo esta en base a la posición de los parámetros; en algunos lenguajes, como Ada, se pueden remplazar explícitamente por el nombre; en el ejemplo anterior: desplazamiento(b=>X, c=>50, a=>R).

Los lenguajes de programación utilizan algunas convenciones para mapear los parámetros actuales a los formales, entre estos: llamadas por nombre, llamadas por referencia, llamadas por valor y llamadas por valor-resultado.

9.5.2. Llamadas por Nombre

Las llamadas por nombre tienen un concepto similar a la expansión de macros en los lenguajes ensambladores; se ejecuta la sustitución del nombre del parámetro formal por el del parámetro actual en el subprograma. No existe evaluación de los parámetros en la transmisión de los mismos.

La técnica básica para implementar llamadas por nombre es la de considerar a los parámetros actuales como subprogramas sin parámetros (llamados “thunks”). Cuando en el programa se refiere al nombre del parámetro formal se lo remplaza ejecutando el “thunk” correspondiente y evaluando el parámetro actual.

FORTRAN, en sus inicios, y Algol 60 adoptaron el método de pase de parámetros por nombre.

9.5.3. Llamadas por Referencia

El método más común de transmisión de parámetros es el de llamadas por referencia; se pasa el l-value del parámetro actual; es decir se remplaza el parámetro formal por un puntero que contiene la dirección del parámetro actual. El objeto de dato no cambia su posición en memoria.

Para la transmisión de las direcciones se genera una lista que es almacenada en un área común de memoria que va a ser compartida con el subprograma. El control se transfiere al subprograma (generándose los registros de activación) y la lista de punteros con las direcciones es utilizada durante la ejecución del subprograma para acceder a los r-values de los parámetros actuales.

Durante la ejecución del subprograma el objeto de dato es leído y modificado cuándo se encuentra en el lado derecho de una asignación.

9.5.4. Llamada por Valor

Si la transmisión del parámetro es el r-value del mismo entonces se está pasando una copia del valor del parámetro actual. El procedimiento de la transmisión de parámetros es similar al de llamada por referencia con la diferencia que no se pasa el l-value sino el r-value.

Al no tener la dirección del objeto de dato, este no puede ser modificado durante la ejecución del subprograma.

9.5.5. Llamada por Valor-Resultado

La transmisión de parámetros por valor-resultado es una composición de la llamada por valor y la llamada por referencia; durante la ejecución del subprograma se utiliza el r-value del parámetro actual y al terminar de ejecutarse el subprograma se usa el l-value del parámetro actual para actualizar el valor de los objetos de datos.

El método de valor-resultado se desarrolló en Algol-W con la finalidad de acelerar la ejecución de los subprogramas al utilizar los parámetros actuales como variables locales al subprograma.

La mayor parte de los lenguajes utilizan dos o más de estas técnicas. El lenguaje C utiliza llamada por valor y por referencia, Ada usa llamadas por valor, por

resultado y por valor-resultado, en el ejemplo a es llamada por resultado, b es llamada por valor-resultado y c es llamada por valor.

Capítulo 10

Administración de Almacenamiento

10.1. Almacenamiento de Pila vs. Almacenamiento Heap

Anteriormente se ha utilizado un modelo simplificado de la memoria de un sistema. En este modelo, una vez cargado el programa objeto las variables tienen una asignación estática por cuánto se les asigna una posición fija en memoria. Se tienen dos punteros, el puntero de pila (stack pointer) y el heap.

Durante la ejecución del programa, cada llamada a una función, procedimiento o rutina, genera un registro de activación y el puntero de pila se desplaza hacia abajo; una vez que se termina de ejecutar la rutina, el registro de activación se elimina y el puntero de pila se desplaza hacia arriba. El manejo de memoria, en este caso, e sencillo.

La administración de la memoria dinámica (apuntada por el heap) es diferente; cuando, durante la ejecución del programa, se crea una variable se asigna dinámicamente un espacio en la región del heap para almacenar esa variable. En los lenguajes dinámicos todas las variables son creadas de esa manera, en los lenguajes orientados a objetos, cada vez que creamos una instancia con new se

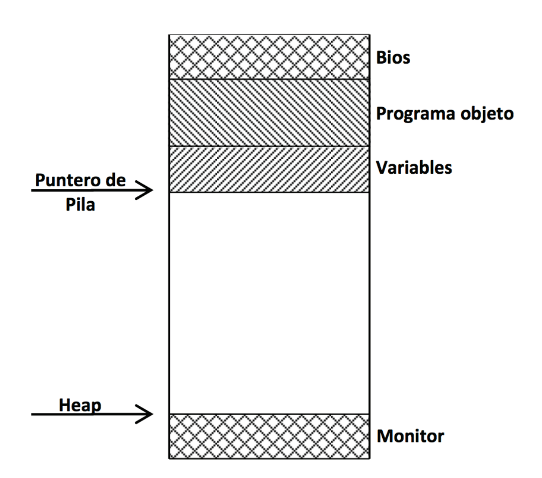


Figura 10.1: Modelo de memoria simplificado

la almacena en el heap y en lenguajes estáticos como C cuando queremos asignar dinámicamente un espacio de, por ejemplo 20 bytes, utilizamos malloc(20) (memory allocation) que retorna un puntero a un bloque de almacenamiento de 20 bytes.

El puntero de pila va bajando a medida que se llaman a los procedimientos; por ejemplo, llamadas recursivas, y el heap va subiendo con las asignaciones dinámicas, reduciéndose el espacio de memoria disponible (entre los punteros de pila y heap) pudiendo el programa pararse por falta de memoria libre.

Así como se asigna memoria dinámicamente, se libera memoria utilizando free, disposal o las variables creadas dinámicamente por un procedimiento que termina de ejecutarse. El proceso de asignación dinámica de memoria y de liberación de la misma crea un problema de fragmentación de memoria, especialmente con asignación de bloques de tamaño fijo.

Si se ejecutan las siguientes instrucciones:

```
allocate(x)
allocate(y)
```

```
allocate(z)
allocate(w)
free(x)
free(z)
allocate(a)
```

se tiene el siguiente almacenamiento:

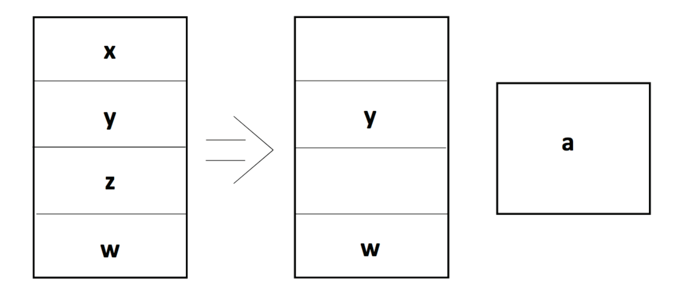


Figura 10.2: Asignación de memoria después de ejecutarse las instrucciones.

El tamaño de a es de 2 bloques, debido a la fragmentación de memoria, no puede ser almacenado a menos que se compacte la memoria

En el siguiente ejemplo:

```
procedure M
   puntero X
   procedure N
      puntero Y
      allocate(Y)
      .
      .
      .
      X=Y
   return
   .
   .
   .
```

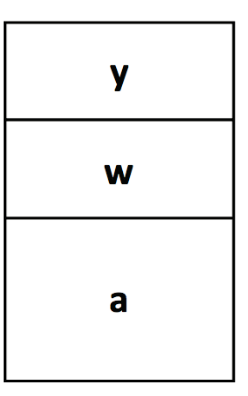


Figura 10.3: Estado de la memoria después de compactación y almacenamiento de a

Al ejecutarse la instrucción X=Y en el procedimiento N, se genera un puntero X que apunta a la misma posición que Y.

Al terminar de ejecutarse el procedimiento N, se elimina el registro de activación con todas sus variables, por lo que el puntero X queda apuntando a una variable que ya no existe y en una posición que el administrador de memoria la declaró libre. Este problema se lo denomina puntero colgante (dangling pointer), el error es difícil de detectar y es una de las razones principales por las que se restringió, en los lenguajes, la utilización de punteros por parte de los programadores.

10.2. Asignación Dinámica de Memoria

La reducción del espacio libre de memoria y la fragmentación de la memoria dinámica hacen necesario que en el heap se detecte la memoria realmente ocupada y que se la compacte eliminando la fragmentación y ampliando la capacidad de memoria disponible. Las técnicas para la gestión de la memoria dinámica puede ser considerada en dos categorías: manejo de bloques de tamaño fijo o de tamaño variable. Las técnicas más simples son las que manejan bloques de tamaño fijo.

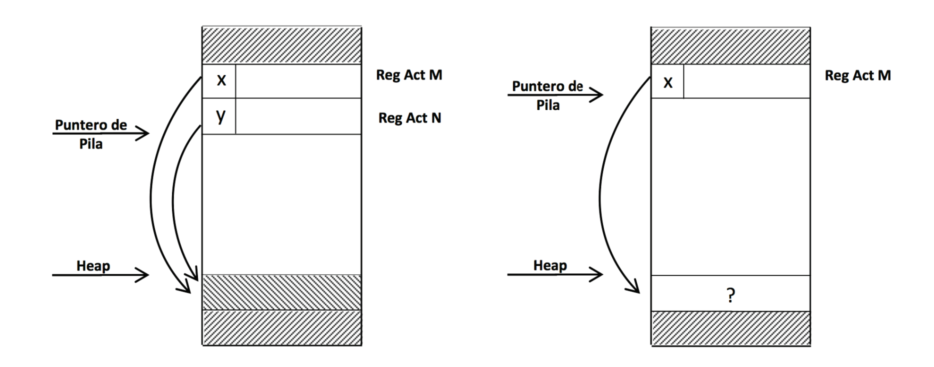


Figura 10.4: (a) Memoria después de X=Y; (b) Memoria al terminar el procedimiento N

10.2.1. Elementos de Tamaño Fijo

La administración de memoria con elementos de tamaño fijo, usualmente es realizada por medio de una lista que enlaza los bloques libres de memoria. Para asignar un bloque, simplemente se libera de la lista el primer bloque y se entrega un puntero al procedimiento que solicitó memoria. Cuando se libera un bloque, entonces se lo incorpora a la lista. Durante este proceso puede suceder el problema del puntero colgante anteriormente descrito o puede suceder lo siguiente:

```
int *p, *q;
...
p=malloc(sizeof(int));
q=malloc(sizeof(int))
p=q;
```

Al asignar a p la dirección a la que apunta q, la memoria que anteriormente estaba asignada a p queda marcada como ocupada pero no existe ningún sendero válido para acceder a ella y se genera basura.

Para poder compactar la memoria es necesario primeramente detectar la memoria ocupada. El recolector de basura (garbage collector) es el sistema que

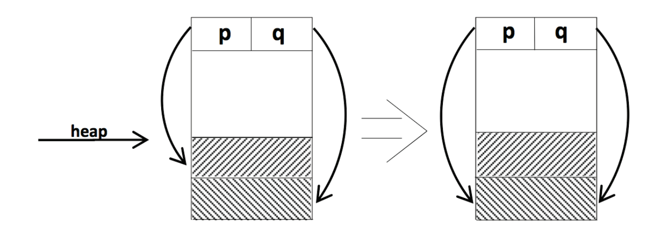


Figura 10.5: Generación de basura

de manera automática encuentra bloques que no están en uso y hace este espacio disponible para ser reutilizado. Un bloque es considerado basura si no es alcanzable, por ningún sendero, por el programa que se está ejecutando.

Existen dos procesos entrelazados, el primero, de detección, que distingue objetos vivos de la basura y el segundo que reclama la basura para poder reutilizarla.

Existen dos técnicas básicas para recolección de basura:

10.2.1.1. Cuenta de Referencias

En el sistema de cuenta de referencia[1], introducido por P. Stygar en 1967, cada objeto tiene un contador asociado que cuenta el número de referencia (punteros) a ese bloque. Cada vez que se crea un nuevo sendero a ese objeto, el contador se incrementa y cada vez que se elimina un sendero, se decrementa.

Cuando un bloque tiene una cuenta de 0 entonces esta libre. Un tema importante con esta técnica es que mientras el programa se ejecuta, los ajustes y el chequeo de los contadores se ejecutan paralelamente, sin parar el programa que se encuentra en ejecución.

Un problema, poco común, que tiene que resolver esta técnica es la ciclicidad de ciertas estructuras:

Si el sendero que apunta a la estructura circular es eliminado, todos los contadores tendrán el valor de 1 pero entonces no se podrá acceder a la estructura.

Un costo de la técnica de cuenta de referencia es que cuando se crea o destruye un puntero hay que ajustar los contadores y cuándo se mueve un puntero de

[1] https://ritdml.rit.edu/bitstream/handle/1850/5112/PWilsonProceedings1992.pdf?sequence=1

Figura 10.6: Cuentas de referencia en la memoria heap

un objeto a otro objeto es necesario ajustar dos contadores lo que ocasiona un retardo en la ejecución de las instrucciones.

10.2.1.2. Marcado y Barrido

Este algoritmo planteado por John McCarthy+ en 1960, como su nombre lo indica, tiene dos fases:

- Distingue objetos vivos de la basura mediante el seguimiento, desde la raíz, de todos los senderos validos y se marcan con una bandera los bloques de memoria a los que los senderos alcanzan.
- Recolección de la basura. Una vez que se han marcado los bloques de memoria se recolecta toda la memoria no marcada y se la asigna a una lista de memoria disponible de la misma forma que en cuenta de referencia.

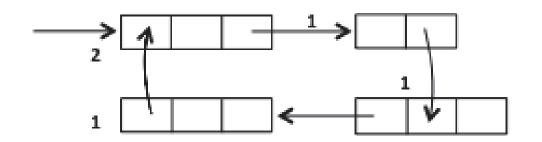


Figura 10.7: Estructura circular

El marcado y barrido tiene dos problemas principales: El costo de la recolección es proporcional al tamaño del heap debido al marcado de los bloques. El segundo problema es que, debido a la fragmentación, se mezclar variables de generaciones diferentes (objetos antiguos y nuevos).

Es importante notar que para ejecutar el proceso de marcado y barrido, el programa debe parar su ejecución.

10.2.2. Recolección Generacional de Basura

Las dos técnicas básicas tienen problemas de costos proporcionales al tamaño del heap debido a los procesos de detectar la basura y compactar la memoria.

Durante la ejecución de un programa, la mayor parte de los objetos tiene un tiempo de vida muy corto mientras un pequeño porcentaje vive mucho más tiempo; de estudios realizados, entre el 80 y el 98 por ciento de las nuevas asignaciones de objetos desaparecen en muy corto tiempo (medido en millones de instrucciones).

Un colector generacional, plantado por C. J. Cheney en 1970, está basado en la organización espacial de la memoria dividida en áreas (semi espacios) para almacenar objetos de diferentes edades o generaciones. Si se divide la memoria en dos áreas: Vieja y Nueva generación, Cuándo se crea un objeto se asigna memoria en el área de Nueva generación hasta que el área se llene; entonces, los objetos vivos pasan al área de Vieja generación. El área de Vieja generación se llenará más lentamente pero eventualmente se llenará y tendrá que recolectarse basura. Smalltalk utiliza este método con 8 generaciones; Squeak utiliza un algoritmo generacional de marcado y barrido con dos generaciones. La máquina virtual de Java (JVM) no define que tipo de algoritmo se debe utilizar; cada implementador de JVM debe decidir que algoritmo utiliza.

10.2.3. Compactación, Pare y Copie

Todos los algoritmos, una vez detectada la basura, necesitan desfragmentar el heap. Existen dos métodos principales:

- Compactación: Los colectores mueven los objetos vivos hacia un extremo del heap por lo que el otro extremo es un área libre contigua. Todas las referencias son actualizadas para que apunten a la nueva posición del objeto. Este algoritmo tiene el costo por acceder y mover cada uno de los objetos.
- Pare y Copie: En este algoritmo (stop and copy) el sistema trabaja con dos memorias heap. Cuando la de trabajo esta llena y fragmentada, se mueven todos los objetos vivos al otro heap y se los coloca uno junto a otro eliminando la fragmentación y convirtiendo a este heap en el de trabajo, los punteros que apuntan a los objetos en el nuevo heap de trabajo quedan almacenados en el heap anterior, el que esta disponible para el siguiente ciclo. El costo asociado es el de utilizar doble capacidad de memoria

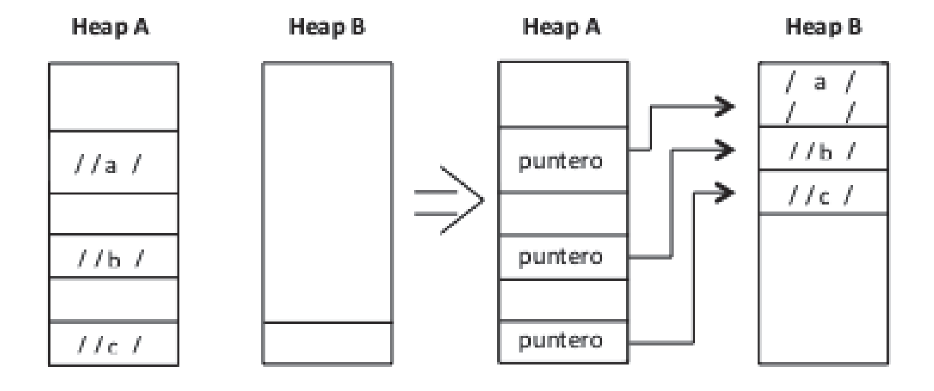


Figura 10.8: Algoritmo Pare y Copie

Los algoritmos de recolección de basura y desfragmentación son importantes. Algunas implementaciones de lenguajes utilizan diferentes algoritmos depen-

diendo del tipo de aplicación y del momento durante la ejecución del programa. Al evaluar un algoritmo de recolección de basura se debe analizar:

- Tiempo de pausa. ¿Cuánto tiempo debe parar el programa?
- Predictibilidad de la pausa. ¿Se puede programar cuando debe parar el programa?
- Uso de CPU. ¿Qué porcentaje del tiempo total de CPU es dedicado a la recolección de basura?
- Memoria Heap. ¿Cuál es el tamaño total de la memoria heap que se va a utilizar?
- Interacción con la memoria virtual. Si el heap interactúa con memoria virtual, ¿el recolector maneja la interacción?
- Impacto en tiempo de corrida. ¿El algoritmo cuánto afecta al tiempo de corrida del programa? El análisis también depende si la programación es concurrente, paralela, distributiva; la performance se ve también afectada si el programa y la recolección de basura se ejecutan en las arquitecturas actuales.

Bibliografía

[1] Anthony Aaby. *Theory: Introduction to Programming Languages*. Self-published, 2004.

[2] Mike Grant, Zachary Palmer, and Scott Smith. *Principles of Programming Languages*. Self-published, 2011.

[3] Robert Harper. *Practical Foundations for Programming Languages*. Carnegie Mellon University, 2012.

[4] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. Brown University, 2007.

[5] Terrence Pratt and Marvin Zelkowitz. *Programming Languages: Design and Implementation*. Prentice Hall, 4th. edition, 2000.

[6] Roberto Rogríguez, Alvaro Prieto, and Encarna Sosa. *Programación Orientada a Objetos*. Self-published, 2004.

[7] Robert W. Sebesta. *Concepts of Progamming Languages*. Pearson, 10th edition, 2012.